Leitprogrammartige Unterrichtsunterlagen zum Thema Digitale Daten

Mentorierte Arbeit

ETH Zürich

Florian Köhl

Konzeption der Unterrichtseinheit

Fachlicher Inhalt

Concept Map

Die folgenden drei Abbildungen zeigen eine erste *grobe Auslegeordnung* meiner Gedanken zum Thema digitale Daten in Form von Concept Maps. Der besseren Übersicht halber stelle ich diese in drei Teilen dar, eine für jeden der drei Teilbereiche Zahlen, Text und Bilder.

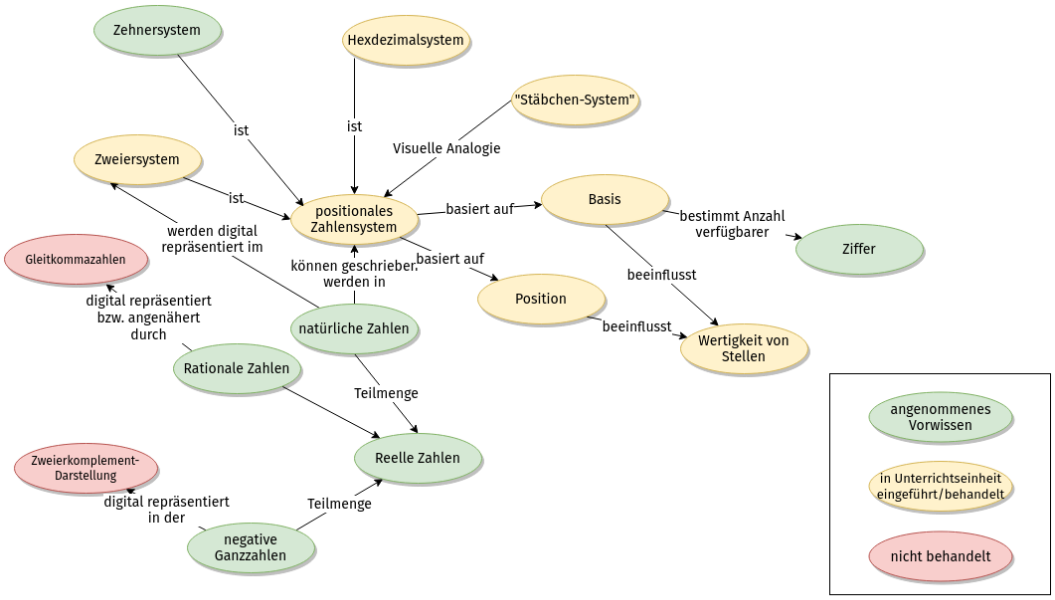


Abbildung 1: Concept Map zum Block *Zahlen(systeme) und deren Darstellung*

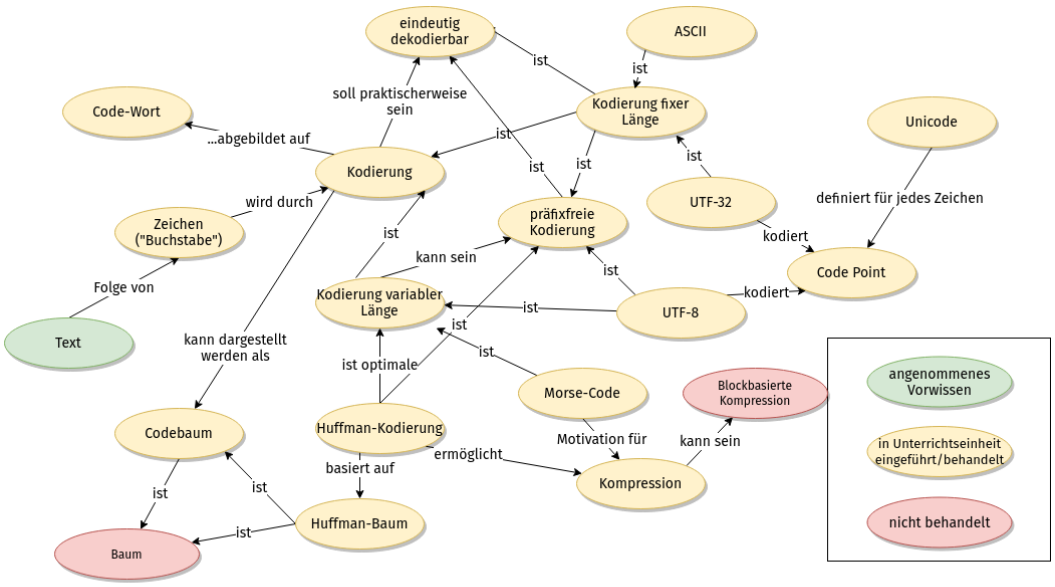


Abbildung 2: Concept Map zum Block *Text/Encoding*

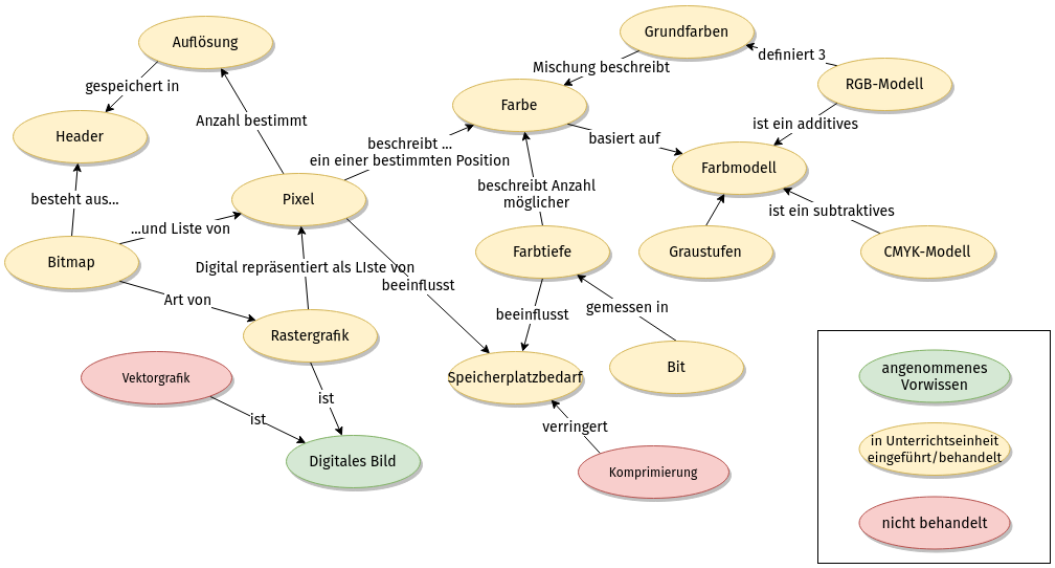


Abbildung 3: Concept Map zum Block *Digitale Bilder*

Vorwissen und Abgrenzung

Diese LPU richten sich an Schülerinnen und Schüler (SuS) im obligatorischen Fach Informatik. In meinem Fall sind die SuS im Schnitt etwa 15 Jahre alt und befinden sich im dritten Jahr des sechsjährigen Langzeitgymnasiums. Im Folgenden gehe ich auf das angenommene Vorwissen der SuS ein.

- Die SuS kennen den Begriff des Algorithmus und haben Beispiele und Anwendungen des Letzteren in verschiedensten Gebieten gelernt und geübt.
- Aus dem Mathematikunterricht sind die SuS mit dem Konzept der Potenzrechnung vertraut.
- Die SuS kennen die Summennotation $\sum$.
- Die SuS kennen die Ganzzahldivision und die entsprechende Restberechnung (Modulo) und die entsprechenden Operatoren in Python.
- Die SuS haben rudimentäre Programmierkenntnisse in der Sprache Python und kennen das Konzept des modularen Aufbaus zum Vereinfachen und Herunterbrechen von Problemen.
- Idealerweise haben die SuS Vorkenntnisse aus dem Bereich der Kombinatorik. Dies ist jedoch keine strikte Voraussetzung und kann gut parallel mit ein paar einfachen Beispielen vertieft werden. Diese LPU liefern die allernötigsten Grundlagen anhand einer Aufgabe (Berechnung der Anzahl möglicher Wörter einer gewissen Länge über dem binären Alphabet), ohne jedoch tiefer darauf einzugehen.

Konzeption und Überlegungen hinter der Gestaltung der LPU

Eins sei vorab erwähnt: der Inhalt dieser LPU ist nicht für eine Doppelstunde gedacht! Sie bieten genügend Material für mindestens 10, lieber aber 12 oder mehr Lektionen. Und das ist nur der Anfang. Der nächste Abschnitt gibt einige Ideen, wie man nach Absolvieren dieser Grundlagen fortfahren kann.

Information und (digitale) Daten, so lautete zu Beginn der Titel dieser Unterrichtsunterlagen. Nach reichlichem Überlegen habe ich mich schlussendlich dazu entschieden, das Wort *Information* aus dem Titel zu streichen. Nicht, weil es nicht interessant oder informativ[1] wäre, sondern da eine Abhandlung dieses Begriffes für das Grundlagenfach schlicht zu kompliziert ist. Informationstheoretische Konzepte wie Entropie oder die Definition von Information als *Beseitigung von Ungewissheit* sind für SuS in diesem Alter schlicht zu abstrakt und mathematisch zu involviert.

Nichtsdestotrotz empfinde ich es als äusserst wichtig, den Unterschied zwischen Information und Daten zu thematisieren. Der Einstieg in diese LPU tut genau das, und zwar auf einer intuitiven Ebene, welche die Idee "rüberbringt", ohne dabei auf mathematische Konzepte zurückzugreifen. Ein treffendes und intuitives Exempel dafür zu finden, war gar nicht so einfach. Mit der Uhrzeit als Information und deren Darstellung hoffe ich aber, ein einleuchtendes Beispiel gefunden zu haben.

Das zweite kleine Kapitel führt die binäre Speichereinheit "Bit" ein, zusammen mit dessen Vielfachen Byte, Kilobyte, Megabyte etc. Ebenfalls findet sich dort ein kleiner Exkurs in die Kombinatorik, um die davon benötigten Grundlagen für die folgenden Abschnitte zu schaffen.

Den Einstieg ins dritte Kapitel bilden positionale Zahlensysteme. Dies geschieht am Beispiel des (hoffentlich!) bekannten Dezimalsystems, und wird darauf aufbauend verallgemeinert. Als Analogie nutze ich hier ein "Stäbchen-System", welches in der Folge zur Einführung des Fünfersystems, später auch des Binärsystems verwendet wird. Dieses kann auch haptisch im Unterricht eingesetzt werden, um die trockene Mathematik etwas aufzulockern. Weiter bildet es die Grundlage eines der beiden vorgestellten Verfahren, um Zahlen in einem beliebigen Zahlensystem darzustellen. Den Abschluss bildet das Hexadezimalsystem, welches in der Welt der Informatik bekanntlich sehr verbreitet ist, und ausserdem "lustige"[2] Zahlen zur Umrechnung bietet. In diesem Kapitel beschränke ich mich bewusst auf die Darstellung positiver ganzer Zahlen. Einer- oder Zweierkomplementdarstellung für negative Zahlen wären sicherlich spannende Anknüpfungspunkte, ebenso die Darstellung bzw. Annäherung von rationalen Zahlen durch Gleitkommaarithmetik, aber diese Themen gehen meiner Meinung nach etwas zu weit für den Einstieg in das Thema.

Das darauf folgende Kapitel widmet sich der digitalen Repräsentation von Text. Ausgehend von der naheliegenden Idee, Zeichen als Zahlen abzubilden und durchzunummerieren wird der Begriff der Kodierung und das Konzept der eindeutigen Dekodierbarkeit eingeführt. Genau so wichtig ist es aber meiner Meinung nach, die Standardisierung hervorzuheben, ohne welche eine Kommunikation in der digitalen Welt nicht möglich wäre. ASCII und Unicode werden hier als Beispiele herbeigezogen, zu Letzterem wird auch die Kodierung der Code Points mittels UTF-32 und UTF-8 angesprochen. Auf die Eindeutigkeit von UTF-8 wird dabei nicht im Detail eingegangen, dies wird aber im nachfolgenden Kapitel nachgeholt.

[1] pun intended

[2] `0xC0FFEE`, `0xACE`, `0xBEEF`, …

Als Motivation für die Einführung von Kodierungen variabler Länge habe ich mich für die Morsekodierung entschieden. Einerseits, da sie bereits die Häufigkeiten der zu kodierenden Zeichen berücksichtigt, und andererseits, weil sie sich gut eignet, um das Konzept des Codebaums einzuführen. Bei dessen Beschreibung habe ich bewusst auf eine formale Definition eines Baumes im graphentheoretischen Sinn verzichtet, und versuche vielmehr, auf visueller Ebene zu arbeiten. Präfixfreie Kodierungen lassen sich dankbarerweise leicht aus dieser Codebaum-Darstellung herleiten. Die Huffman-Kodierung beziehungsweise die Konstruktion eines Huffman-Codebaums bildet den Abschluss dieses Kapitels.

Der letzte Teil dieser LPU widmet sich digitalen Bildern. Das Vorgehen, welches sich auch in der Praxis als gut durchführbar erwiesen hat, basiert auf einer vereinfachten Version des Bitmap-Formats, bei dem der Header lediglich aus zwei Binärzahlen (Dimensionen des Bildes), der Payload aus einer Liste von Pixelwerten als besteht. Den Anfang bilden monochrome Bitmaps, anhand welcher die Speicherstruktur demonstriert wird. Aufbauend darauf werden Graustufenbilder eingeführt, wodurch auch das Konzept der Farbtiefe ein Thema wird. Ein kleines bisschen Farbentheorie (additive und subtraktive Farbmodelle) darf natürlich vor der Einführung von RGB-Bildern nicht fehlen. Anhand letzterer wird nochmals der Begriff der Farbtiefe thematisiert und graphisch aufgezeit, welchen Einfluss diese auf das Aussehen eines Bildes bzw. die verfügbaren Farben hat.

Lösungen zu ausgewählten Aufgaben runden diese LPU schlussendlich ab.

Ein paar Worte zum Stil der Arbeit

Ich habe versucht, den Inhalt so formal wie nötig und so anschaulich wie möglich zu gestalten. Wie immer finden sich viele Beispiele und noch mehr Aufgaben zu allen Themen. Auch wenn dies auf den ersten Blick abschreckend wirken mag: Die Meinung ist nicht, dass jede Aufgabe vollständig gelöst werden soll. Einige oder Teile davon können während dem Unterricht gelöst werden, andere als Hausaufgaben, wiederum andere als Vorbereitung auf eine Prüfung.

Mit Hilfe von kurzen Denkanstössen sollen die SuS zum aktiven Mitdenken ermuntert werden. Diese als ?-Kästchen dargestellten Zwischenfragen werden meistens bald darauf in den LPU beantwortet, wodurch die SuS ihre Theorien bestätigen oder hinterfragen können. Ebenfalls geben sie der Lehrperson eine gute Gelegenheit, diese als interaktive Elemente im Unterricht selbst einzubauen.

Weiter habe ich mir Mühe gegeben, zwischendurch ein paar Auflockerungen einzubauen. Ein binäres Kreuzworträtsel, ein binäres Sudoku, eine ”Fun Facts” Box rund ums Thema Emoji, Links zu weiterführenden interessanten Videos o.ä. Solche Elemente bieten gern gesehene Abwechslung und rücken den manchmal trocken erscheinenden Stoff in ein anderes, hoffentlich besseres Licht.

Bilder und deren Herkunft

Verwendete Bilder in dieser LPU sind, wo nötig, mit Quellenangaben versehen. Wenn keine Quellenangabe ersichtlich ist, so stammen die Bilder aus der freien Bildersammlung Pixabay und fallen unter deren Lizenz (Verwendung und Bearbeitung ohne Quellenangabe erlaubt).

Weiterführende Inhalte und Anknüpfungspunkte

Diese LPU bieten die Grundlagen für (zu) viele interessante Folgethemen! Im Folgenden gebe ich ein paar Ideen für Unterrichtsblöcke oder Projekte, welche ich mit meinen Klassen ausprobiert habe.

- **Bildbearbeitung mit Python**: Einfache Bild-Filter (ja, so nennt man das heutzutage) zu schreiben ist mit Python gar nicht so schwierig. Mit dem Modul pillow und einer kleinen Helper-Datei zum Laden von Bildern oder Pixellisten sind die SuS sehr wohl in der Lage, einfache Farbinversion vorzunehmen. Ob Graustufen, Farbinvertierung, Spiegelungen, oder Sepia-Effekt: Es gibt hier unzählige konzeptionell simple, aber visuell eindrückliche kleine Farb- und Bildmanipulationen, die einfach umsetzbar sind. Ein ideales Mittel, um for-Schleifen und (zweidimensionale) Listen sowie deren Indexierung zu behandeln oder zu üben!

  Für SuS im Ergänzungsfach wären auch einfache Kernel-Filter wie etwa ein Kantenfinder oder Blur-Effekt denkbar.

- **Bildkomprimierung**: Einfache Komprimierungsverfahren wie Farbindizierung oder Runlength Encoding können im Anschluss gut unterrichtet werden. Für das Ergänzungsfach wären auch verlustbehaftete Verfahren wie JPEG eine Option, obwohl da einiges an mathematischen Vorkenntnissen benötigt wird, wenn man ins Detail gehen will…Aber spannend ist es allemal!

- **Fehlererkennung und -Korrektur**: Einfache fehlererkennende Konzepte wie beispielsweise Kreditkartennummern oder Barcodes kommen sehr gut an und bilden gute Anknüpfungspunkte. Auch fehlerkorrigierende Codes (Hamming!) sind gut und dankbar umsetzbar, sofern man auf einer einigermassen hohen Ebene bleibt und informationstheoretische Details weglässt.

- **Vektorgrafiken**: Bringt man den SuS im Anschluss den Aufbau und die Grundlagen von SVG-Grafiken bei und lässt sie damit beispielsweise das Instagram-Logo[3] nachbauen, ist das Wort *Pause* plötzlich ein Fremdwort.

3

Lernziele nach dem Zielebenenmodell

Leitidee

Das Verständnis, wie Daten digital in binärer Form gespeichert oder repräsentiert werden, stellt einen fundamentalen Grundbaustein der Informatik und des algorithmischen Denkens dar. Nicht nur bildet es die Grundlage für zahlreiche darauf aufbauende Themengebiete wie das wissenschaftliche Rechnen, Komprimierungsverfahren oder Visual Computing (um nur einige zu nennen), es fördert auch das Abstraktionsvermögen, das Denken in und Entwickeln von Modellen sowie den Fokus auf das Wesentliche, und verdeutlicht schlussendlich auch die Wichtigkeit der Standardisierung. Weiter streift es wichtige und interessante Themen aus unterschiedlichsten Disziplinen wie der Farbenlehre, der Informations- oder Kodierungstheorie.

Dispositionsziele

- Die SuS haben eine intuitive Vorstellung des Konzepts Information und verstehen Daten als deren Darstellung.
- Die SuS kennen das Bit als kleinste digitale Speichereinheit.
- Die SuS realisieren, dass alle digitale Daten schlussendlich als Folge von Bits gespeichert sind.
- Die SuS kennen verschiedene Darstellungsmöglichkeiten für Zahlenwerte.
- Die SuS erkennen die Wichtigkeit von (Kommunikations-) Standards.
- Die SuS haben eine Vorstellung vom Speicherlayout digitaler Rasterbilder.
- Die SuS schreiben mit Freude eigene einfache Bild- und Farbmanipulationsprogramme für Rasterbilder.

Operationalisierte Lernziele

Die folgende Tabelle listet die operationalisierten Lernziele dieser LPU auf.

Lernziel	Kontrolle (Bsp.)
Die SuS beschreiben den Unterschied und den Zusammenhang zwischen Information und Daten.	
Die SuS nennen Beispiele digitaler Daten.	
Die SuS rechnen zwischen Bits und Bytes sowie deren SI-Vielfachen um.	A3
Die SuS rechnen Dezimalzahlen ins Binär-, Hexadezimal- oder ein beliebiges anderes positionales Zahlensystem um.	A5, A8, A14, …
Die SuS rechnen Zahlen von einem beliebigen positionalen Zahlensystem ins Zehnersystem um.	A6, A15, …
Die SuS wissen, an was man eine gerade bzw. ungerade Binärzahl erkennt.	A10
Die SuS addieren zwei Binärzahlen schriftlich.	A13
Die SuS kennen die Begriffe *Kodierung*, *Code-Wort*, *kodieren* und *dekodieren* und können diese definieren oder anhand eines Beispiels erklären.	
Die SuS geben ein Beispiel einer Kodierung fixer Länge und begründen, weshalb diese eindeutig dekodierbar ist.	A18
Die SuS wandeln, gegeben eine ASCII-Tabelle, einen englischen Text in die entsprechende ASCII-Bitfolge um.	A21
Die SuS übersetzen, gegeben eine ASCII-Tabelle, einen mit ASCII kodierten Text von der resultierenden Bitfolge zurück in den Originaltext.	A23
Die SuS kennen Unicode als kanonische Sammlung aller je verwendeten Schriftzeichen.	
Die SuS kodieren eine Zeichenfolge, bei welcher die Unicode Code-Points der einzelnen Zeichen angegeben sind, mittels UTF-32 oder UTF-8.	A25, A26
Die SuS erklären ihren jüngeren Geschwistern, dass Emojis 'nur' Zeichen sind, die ebenfalls durch das Unicode-Konsortium verwaltet werden.	

Tabelle 1: Operationalisierte Lernziele der LPU

Lernziel	Kontrolle (Bsp.)
Die SuS begründen, weshalb die binäre Morsekodierung nicht eindeutig dekodierbar ist.	
Die SuS stellen eine gegebene Kodierung als Codebaum dar.	
Die SuS definieren den Begriff *Präfix* und zählen alle möglichen Präfixe eines (kurzen) Wortes auf.	A27
Die SuS erklären mit Verweis auf den Codebaum, weshalb eine präfixfreie Kodierung eindeutig ist.	A28, A31
Die SuS erkennen, ob eine gegebene Kodierung präfixfrei ist.	A29
Die SuS bauen den Huffman-Baum zu einer gegebenen Zeichenfolge Schritt für Schritt auf.	A33
Die SuS leiten aus einem Huffman-Baum die Huffman-Kodierung her und kodieren ein gegebenes Wort damit.	A33
Die SuS wissen, dass ein Pixel nicht einfach ein kleines farbiges Quadrat ist.	
Die SuS berechnen die Auflösung eines Bildes gegeben dessen Dimensionen.	A35
Die SuS wissen, wie eine monochrome Bitmap aufgebaut ist und wie diese digital im Arbeitsspeicher abgelegt ist.	A37
Die SuS erklären, weshalb der Header einer (vereinfachten) Bitmap-Datei zum Darstellen einer solchen benötigt wird.	A36, A37
Die SuS skizzieren eine als Bitfolge gegebene Bitmap auf einem Blatt Papier.	A37
Die SuS erklären den Begriff *Farbtiefe* anhand eines Beispiels.	
Die SuS erläutern in wenigen Sätzen, wie verschiedene Graustufen in Bitmaps dargestellt werden.	
Die SuS beschreiben die Auswirkung der Farbtiefe auf die darstellbaren Farben.	A38, A40
Die SuS erklären den Unterschied zwischen additiven und subtraktiven Farbmodellen und nennen je ein Beispiel.	A39
Die SuS zählen die Grundfarben des RGB-Farbmodells auf und nennen die drei primären Mischfarben.	A39, A43
Die SuS berechnen den Speicherplatzbedarf einer Graustufen- oder RGB-Bitmap gegeben Auflösung und Farbtiefe des Bildes.	A41
Die SuS erkennen in einem manipulierten Farbbild, welcher der drei Farbkanäle weggelassen wurde und begründen ihre Antwort.	A42

Tabelle 2: Operationalisierte Lernziele der LPU (ctd)

Literaturverzeichnis

[1] Juraj Hromkovič: einfach INFORMATIK - Daten darstellen, verschlüsseln, komprimieren, Sekundarstufe 1. Klett und Balmer Verlag, 1.Auflage, 2018.

[2] T. Ottmann & P. Widmayer: Algorithmen und Datenstrukturen. Spektrum Akademischer Verlag, 5. Auflage, 2012.

[3] Jens Gallenbacher: Abenteuer Informatik. Springer Verlag, 4. Auflage, 2017.

[4] Oliver Probst: Kürzeste Wege. Leitprogrammartige Unterrichtsunterlagen, 2015.
http://www.abz.inf.ethz.ch/maturitatsschulen/unterrichtsmaterialien/
Abgerufen am 4.2.2021

[5] Numberphile: Base 12 [Videodatei], 12. Dezember 2012,
https://youtu.be/U6xJfP7-HCc
Abgerufen am 13.10.2021

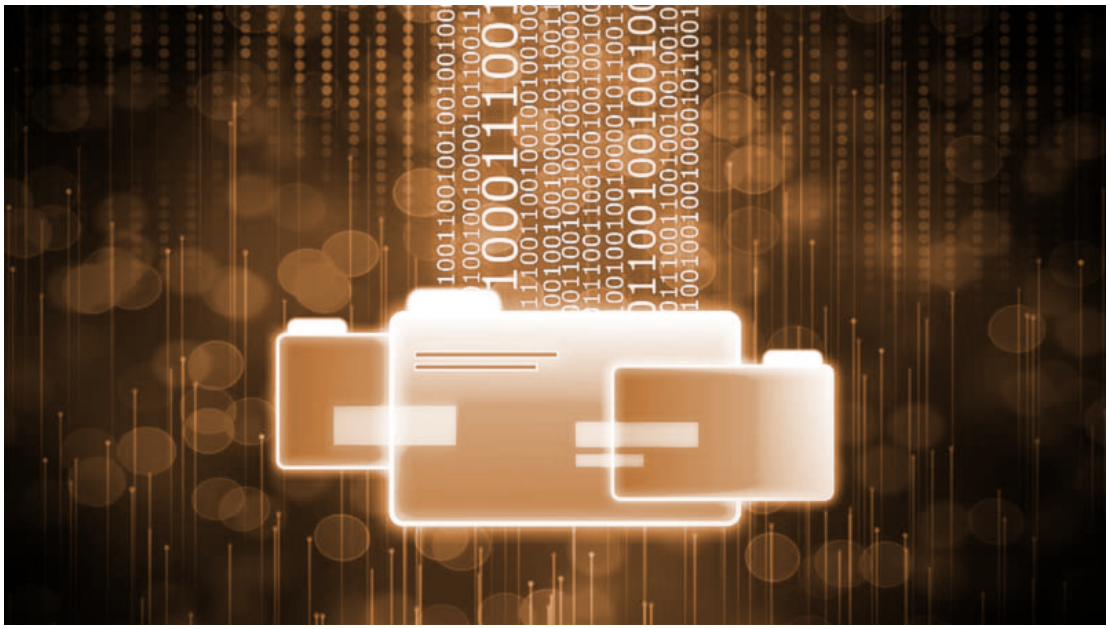
[6] Characters, Symbols and the Unicode Miracle - Computerphile [Videodatei], 20. September 2013,
https://youtu.be/MijmeoH9LT4
Abgerufen am 13.10.2021

[7] Emoji and the Levitating Businessman - Computerphile [Videodatei], 9. Juli 2014,
https://youtu.be/tITwM5GDIAI
Abgerufen am 13.10.2021

[8] Digital Images - Computerphile [Videodatei], 15. Februar 2015,
https://youtu.be/06OHflWNCOE
Abgerufen am 13.10.2021

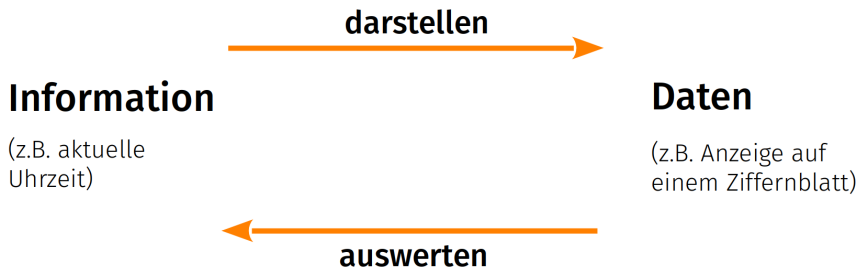
Digitale Daten

Information, Kodierung, Darstellung



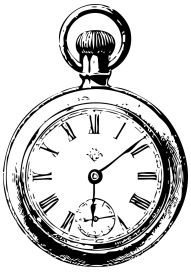
Information und Daten

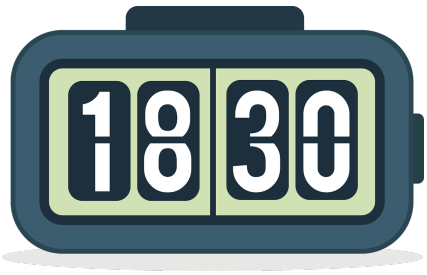
Unter **Information** verstehen wir ganz abstrakt gesagt etwas, welches unser Wissen erweitert. Wenn wir jemandem Information vermitteln oder mitteilen möchten, so müssen wir dies in einer geeigneten Form tun, wir müssen Information also irgendwie **darstellen** können. Je nach Art der Information, die wir vermitteln möchten, können wir das mit unterschiedlichsten Arten von Daten tun: **Zahlen**, **Texte** oder **Bilder**, …; das sind alles Beispiele von Daten. Aus diesen kann durch geeignete Interpretation oder durch **Auswerten** Information gewonnen werden.

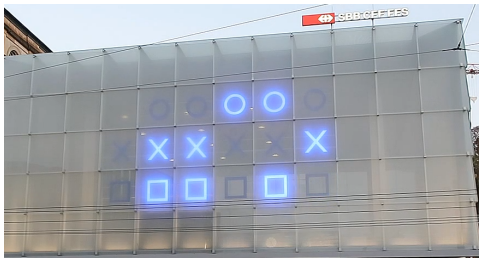


Beispiel – Uhrzeit

Wie würdest du jemandem mitteilen, wie spät es gerade ist? Sicher findest du viele verschiedene Möglichkeiten, die aktuelle Uhrzeit auszudrücken oder darzustellen. Ein paar davon sind durch die folgenden Bilder gegeben:







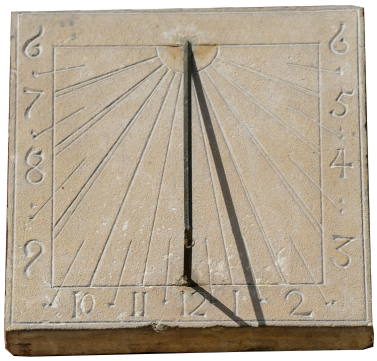


Bild unten links: https://www.flickr.com/photos/kecko/41606615071
Lizenz: CC BY 2.0 (https://creativecommons.org/licenses/by/2.0/)

Aufgabe 1 – Uhrzeiten auswerten

> Kannst du bei allen Beispielen oben die Uhrzeit ablesen? Wie spät ist es auf den Bildern jeweils?

Für den Begriff Information gibt es je nach Blickwinkel ganz unterschiedliche Definitionen. Wir versuchen daher gar nicht erst, dieses Konzept formal zu definieren, sondern arbeiten mit einem intuitiven Ansatz.

Es ist aber wichtig, dass du dir dem Unterschied zwischen Daten und Information bewusst bist und verstehst, wie diese beiden Begriffe zusammenhängen. Oft wird der Begriff Information nämlich synonym mit Daten verwendet. Das kann je nach Anwendungsfall auch Sinn ergeben, wir versuchen hier aber die beiden Konzepte voneinander zu trennen.

Wir wollen uns hier also nicht weiter mit dem (höchst philosophischen) Thema der Information auseinandersetzen, sondern konzentrieren uns primär auf ihre Darstellung in Form von Daten. Genauer gesagt wollen wir uns anschauen, wie wir diese Daten auf einem Computer abspeichern und repräsentieren können. Wir tun dies anhand von drei Arten von Daten:

Zahlen kennen wir alle. Doch deren Darstellung kann, je nach Darstellungsart, komplett anders aussehen. Wir kennen alle das Dezimalsystem oder die Dezimaldarstellung von Zahlen mit ihren Einern, Zehnern, Hundertern und so weiter. Die Zahl 'sieben' stellen wir also als 7 dar. Die Römer hätten diese Zahl aber als VII dargestellt - ein Fünfer (V) gefolgt von zwei Einern (II). Wenn du eine Strichliste führst, so schreibst du diese Zahl wahrscheinlich als 𝍸 II. Eine Zahl – oder besser ihr Wert – kann also auf unterschiedlichste Arten dargestellt werden.

Texte repräsentieren wir ein wenig anders. So stellen wir im deutschsprachigen Raum Texte mit Buchstaben dar. Dazu verwenden wir das uns bekannte Alphabet (A-Z, a-z) sowie ein paar Sonderzeichen (ä,ö,ü,…). Andere Sprachen verwenden leicht andere Alphabete. So finden wir im norwegischen und dänischen Alphabet beispielsweise die Zeichen Ä, ø und å. In anderen Kulturen arbeitet man statt mit Buchstaben mit Schriftzeichen. So bezeichnet das chinesische Zeichen 木 einen Baum. Daraus leiten sich dann die Zeichen 林 (etwa 'Baumgruppe') und 森 (viele Bäume = Wald) ab. Du siehst also, dass Text überall auf der Welt unterschiedlich dargestellt und auch interpretiert wird.

Bilder sind eine dritte Art von Daten, die wir alle kennen. Auch kennst du wohl verschiedenste Darstellungsarten. So kannst du mit Bleistift etwas auf Notizpapier zeichnen, oder mit Wasserfarbe und entsprechendem Werkzeug eine Leinwand bepinseln. Wenn du hingegen etwas ausdruckst, besteht ein Bild hingegen aus vielen kleinen Farbtupfern.

Daten kommen also in ganz unterschiedlichen Formen (und Farben) daher. Doch wie speichern oder repräsentieren wir solche Daten auf dem Computer? Moderne Computer arbeiten nämlich mit einem ganz simplen Alphabet, in dem es nur zwei Ziffern gibt: 0 und 1. Sie kennen von Haus aus keine Dezimalzahlen, auch Schriftzeichen oder geschweige denn Farbtupfern sind ihnen zunächst unbekannt. Alles was sie kennen, sind **Nullen und Einsen**.

In den folgenden Kapiteln wollen wir uns anschauen, wie wir mit diesem äussert beschränkten Alphabet verschiedene Arten von Daten repräsentieren können!

Bits und Bytes - das Computeralphabet

Moderne Computer arbeiten mit einem **binären Alphabet**, das nur aus zwei Ziffern besteht: 0 und 1. Weshalb arbeiten arbeiten Computer mit einem derart beschränkten Alphabet? Wäre es nicht einfacher und intuitiver, wenn wir – wie wir Menschen auch – alle Ziffern von 0-9 verwenden könnten? Einfacher eventuell, für uns Nutzer, aber definitiv nicht für diejenigen Leute, die dann solche Computer bauen müssten.

Computer bestehen im Innersten nämlich aus ganz vielen elektronischen Schaltungen und Bauteilen, durch die Strom fliessen kann. Wenn wir hier zehn verschiedene Zustände unterscheiden möchten, so müssten wir mit 10 verschiedenen Spannungsleveln arbeiten. Dies ist in der Praxis aber nicht ganz einfach umzusetzen, und vor allem sehr fehleranfällig, da die Spannung nicht immer ganz genau reguliert werden kann, oder eine solche genaue Regulation viel zu teuer wäre. Bei zwei verschiedenen Zuständen ist das hingegen ganz einfach umzusetzen. Wenn keine (oder nur ganz wenig) Spannung anliegt, so interpretiert ein Computer diesen Zustand als eine 0. Ab einem gewissen Grenzwert (beispielsweise 3.5 Volt) wird der Zustand als 1 aufgefasst. Wenn wir Information auf einem Computer speichern möchten, müssen wir diese also irgendwie als Folge von Nullen und Einsen hinterlegen.

Bits und Bytes

Die kleinste binäre Informationseinheit ist das so genannte **Bit** (engl: kurz für *binary digit*): Es besteht aus genau einer binären Ziffer, kann also die Werte 0 oder 1 annehmen. Eine Aneinanderreihung von mehreren solchen Bits nennen wir **Bitfolge**.

8 solche Bits hintereinander nennt man ein **Byte**. Ein **Byte** besteht also aus 8 **Bits**. Die Bitfolgen `01101010` und `00001111` sind beides Beispiele von Bytes, da sie beide 8 Bits lang sind.

Wie viele verschiedene Bytes gibt es? Um diese Frage zu beantworten, greifen wir auf die Mathematik, genauer gesagt auf die Kombinatorik zurück! Jedes der 8 Bits kann 2 verschiedene Zustände abbilden, jeweils 0 oder 1. Beginnen wir bei der letzten Stelle. Hier haben wir 2 verschiedene Möglichkeiten. Wenn wir die zweitletzte Stelle betrachten, können wir dort wiederum entweder eine 0 oder eine 1 einsetzen, und für jede dieser Optionen gibt es wiederum 2 Optionen für die letzte Stelle, insgesamt also $2 \cdot 2 = 4$ mögliche Bitfolgen der Länge 2. Für das drittletzte Bit haben wir wieder 2 Möglichkeiten, und multiplizieren diese mit den Möglichkeiten für die letzten 2 Stellen: $2 \cdot 4 = 2 \cdot 2 \cdot 2 = 8$ mögliche Kombinationen. Wenn wir dies nun durchspielen, bis wir bei 8 Bits angekommen sind, erhalten wir

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$

verschiedene Kombinationsmöglichkeiten für eine Bitfolge der Länge 8, oder eben ein Byte.

Die nächstgrösste Einheit nach dem Byte ist das Kilobyte, welches aus 1000 Bytes besteht. Diese und die folgenden Benennungen und Einheiten richten sich nach dem SI-Einheitensystem, welches du sicherlich aus dem Physik- oder Chemieunterricht kennst:

Präfix	Bezeichnung	Abkürzung	Wert
kilo	Kilobyte	kB	1000 Bytes
mega	Megabyte	MB	1000^2 Bytes
giga	Gigabyte	GB	1000^3 Bytes
tera	Terabyte	TB	1000^4 Bytes
peta	Petabyte	PB	1000^5 Bytes

Beispiel – Wieviele Bits?

Wie viele Bits sind in einem Kilobyte enthalten? Dafür rechnen wir die Anzahl Bytes in einem Kilobyte mal die Anzahl Bits in einem Byte. Das ergibt dann

$$1000 \cdot 8 = 8000$$

Bits in einem Kilobyte. In einem Megabyte haben wir folglich also

$$1000 \cdot 1000 \cdot 8 = 1000^2 \cdot 8 = 8000000$$

Bits, also 8 Millionen Bits, 8 Millionen Nullen oder Einsen!

Aufgabe 2 – Ein bisschen Kombinatorik

Wieviele verschiedene Bitfolgen der Länge 2 gibt es? Zähle alle auf.
Wiederhole dies für die Längen 3 und 4.
Versuche danach eine allgemeine Formel für die Anzahl verschiedener Bitfolgen der Länge n zu finden!

Aufgabe 3 – Mega Giga Haga

Wie viele Bits enthält ein Gigabyte?
Wieviele Kilobyte sind in einem Terabyte enthalten?
Wieviele Gigabyte ergeben ein Petabyte?

Zahlen darstellen - Zahlensysteme und die Binärdarstellung

There are only 10 kinds of people in the world.
Those who understand binary and those who don't.

Das **Dezimalsystem** kennst du bestimmt. Eine Zahl, welches in diesem Zahlensystem geschrieben ist, setzt sich – von rechts nach links gelesen – aus Einern, Zehnern, Hundertern, Tausendern usw. zusammen. Jede dieser Stellen ist durch eine Ziffer von 0 bis 9 beschrieben. Um den Wert der Zahl zu bestimmen, rechnen wir jeweils die Anzahl an einer Stelle mal deren Wertigkeit, und addieren diese Zwischenergebnisse dann zusammen.

Beispiel – Dezimalsystem

Betrachte die Dezimalzahl 3172. Sie besteht aus 2 Einern, 7 Zehnern, einem Hunderter und 3 Tausendern. Der Wert ergibt sich also als

$$3 \cdot 1000 + 1 \cdot 100 + 7 \cdot 10 + 2 \cdot 1 = 3172$$

Im Zentrum steht beim Dezimalsystems die Zahl **10**. Sie bestimmt einerseits, **wie viele Ziffern** wir pro Stelle zur Verfügung haben. Andererseits bestimmt sie auch die Wertigkeit der einzelnen Stellen. Diese sind nämlich durch **Potenzen der Zahl** 10 gegeben. So hat die Stelle ganz rechts im Dezimalsystem die Wertigkeit 1, oder anders ausgedrückt 10^0. Die zweite Stelle – also die Zehner – haben die Wertigkeit 10^1, während die dritte Stelle (die Hunderter) die Wertigkeit 10^2 haben. Jede Stelle ist also 10 mal mehr 'Wert' als die Stelle zu ihrer Rechten.

Allgemein können wir eine vierstellige Dezimalzahl als $d_3 d_2 d_1 d_0$ schreiben. d_0 bezeichnet die Stelle ganz rechts, d_1 die zweite Stelle von rechts, usw. Der Wert einer solchen Zahl ist dann also

$$d_3 \cdot 1000 + d_2 \cdot 100 + d_1 \cdot 10 + d_0 \cdot 1 = d_3 \cdot 10^3 + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$$

Im Beispiel oben haben wir also $d_3 = 3$, $d_2 = 1$, $d_1 = 7$ und $d_0 = 2$. Wenn wir diese Zahlen in die Formel oben einsetzen, erhalten wir wieder das gleiche Resultat.

Wir können dies auch allgemeiner formulieren. Der Wert einer n-stelligen Dezimalzahl, welche durch ihre Stellen $d_{n-1} d_{n-2} \ldots d_2 d_1 d_0$ gegeben ist, können wir als

$$\sum_{i=0}^{n-1} d_i \cdot 10^i$$

schreiben. Das auf den ersten Blick abschreckend aus, entspricht aber genau dem, was wir oben beschrieben haben. d_0 steht für die Anzahl Einer. Diese werden mit 10^0, also 1 multipliziert. Dazu zählen wir die Anzahl Zehner (d_1), die wir zuerst noch mit $10^1 = 10$ multiplizieren. Dannach wir die Hunderter (d_2 multipliziert mit $10^2 = 100$ usw.

Zahlen mit Stäbchen ausdrücken

Wir wollen dieses System nun mit einer eigenen, graphischen Darstellung visualisieren. Stell dir vor du hast mehrere Behälter vor dir. In jedem Behälter befinden sich Stäbchen einer gewissen Länge. Der erste Behälter enthält Stäbchen, welche genau 1 cm lang sind. Im zweiten Behälter findest du Stäbchen der Länge 10 cm, im dritten haben alle Stäbchen die Länge 100 cm usw. [1] Wir können nun den Wert einer Zahl darstellen, indem wir diese Stäbchen aneinanderreihen. Die Länge aller verwendeten Stäbchen zusammen repräsentiert dann den Wert der Zahl, die wir gelegt haben.

Wir müssen uns aber auch hier an ein paar Grundregeln halten. So dürfen wir zum Beispiel maximal 9 Stäbchen einer gewissen Länge verwenden. (Die Zahl 12 dürfen wir also nicht aus zwölf 1 cm Stäbchen bilden, sondern aus einem Zehnerstab und zwei Einerstäbchen. Des Weiteren nehmen wir uns vor, die längsten Stäbchen jeweils am Anfang zu legen.

Beispiel – Dezimalstäbchen

Wir wollen die Zahl 27 mit unseren Stäbchen legen. Dafür nehmen wir also 2 Zehnerstäbchen und 7 Einerstäbchen:

Nun ja, das war zugegebenermassen ziemlich einfach. Etwas schwieriger wird es dann, wenn du nicht genau weisst, welche Zahl du legen sollst, sondern du einfach eine vorgegebene Länge genau mit den Stäbchen abdecken sollst – unter Einhaltung unserer Grundregeln natürlich. Stell dir also vor, du hast vor dir auf einem Blatt Papier einen Strich einer bestimmten Länge. Du siehst aber nicht von Auge, wie lange dieser genau ist. Findest du das mit Hilfe der Stäbchen heraus?

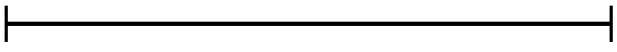
Aufgabe 4 – Stäbchen legen

Beschreibe ein Vorgehen, wie du mit Hilfe unserer Dezimalstäbchen – und unter Einhalten der Grundregeln – die Länge eines Striches bestimmen kannst.

Die Strategie, welche wir für dieses Problem anwenden können, funktioniert wir folgt. Als Erstes suchen wir das längste Stäbchen, welches noch in die abzudeckende Strecke reinpasst, also nicht länger ist als der Strich. Von dieser Sorte nehmen wir dann so viele Stäbchen, bis keines mehr Platz hat. Und für die restliche abzudeckende Strecke wiederholen wir den Vorgang mit den restlichen (kürzeren) Stäbchen.

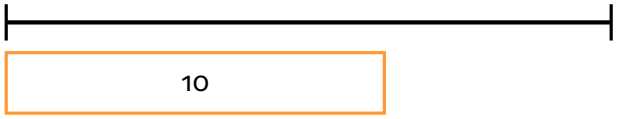
Beispiel – Strecken abdecken

Wir wollen mit unseren Stäbchen folgende Strecke abdecken:

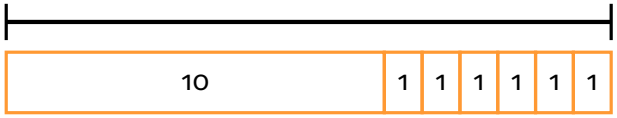


[1] Wir können diese Stäbchen hier leider nicht in Originalgrösse abbilden, da diese sonst nicht auf eine Seite passen würden. Wir stellen die Stäbchen deshalb etwas kürzer dar, geben aber jeweils ihre Länge als Zahl an.

Ein Hunderterstäbchen hat darin definitiv nicht Platz, also versuchen wir es mit den nächstkleineren, den Zehnern. Davon hat nur eines Platz, mit zwei würden wir über das Ziel hinaus schiessen:



Alles, was jetzt noch zu tun bleibt, ist das das Auffüllen mit Einern. Davon haben 6 Platz, die gesuchte Strecke ist also $1 \cdot 10 + 6 \cdot 1 = 16$ cm lang:



Sicherlich siehst du den Zusammenhang zwischen dieser Stäbchendarstellung und unserer gewohnten Zahlennotation aus dem Alltag. Die Anzahl Einerstäbchen entspricht den Einern, die der Zehnerstäben den Zehnern, usw. Der Wert der Zahl erhalten wir dann, in dem wir diese einzelnen Anzahlen mit ihrer Wertigkeit – also der Länge der Stäbchen in der Zahlendarstellung oder der Potenz in der Dezimalschreibweise – multiplizieren und diese Zwischenergebnisse anschliessend aufsummieren.

Das Fünfersystem – (nur) mit links

Wieso zählen wir heutzutage eigentlich (meist) in der Basis 10? Die Antwort, die man darauf am häufigsten findet, hat mit der menschlichen Anatomie zu tun. Wir haben 10 Finger, und diese Finger werden zum (Auf)Zählen verwendet[2]. Was wäre jedoch, wenn wir nur mit einer Hand zählen würden? Also statt dem Dezimalsystem (Zehnersystem) ein Fünfersystem verwenden würden? Wir wollen das im Folgenden kurz untersuchen.

Das Fünfersystem verwendet im Gegensatz zum Dezimalsystem als **Basis** die Zahl **5**. Das wirkt sich sowohl auf die Anzahl unterschiedlicher Ziffern oder Zeichen aus, die wir nützen können, beeinflusst aber auch die Werte der einzelnen Stellen.

Der erste Unterschied ist, dass wir im Fünfersystem nur 5 Ziffern zur Verfügung haben: 0, 1, 2. 3 und 4. Die ersten 5 Zahlen (wenn wir von 0 an beginnen) sind also stellen wir also gleich dar wie im Dezimalsystem. Nach der 4 kommt im Fünfersystem allerdings die 10: Da wir keine weiteren Ziffern mehr haben, müssen wir mit der nächsten Stelle arbeiten. Die zweite Stelle (von rechts) bezeichnet die *Fünfer*, oder anders ausgedrückt die Basis hoch eins (5^1). Die nächste Stelle steht dann für die 25er, also die Basis im Quadrat, gefolgt von den 125ern – die Basis hoch drei (5^3).

Aufgabe 5 – Fünferstäbchen

(a) Wir wollen unsere Stäbchennotation auch für das Fünfersystem nutzen. Welche Arten bzw. Längen von Stäbchen hast du dabei zur Verfügung? Wie viele von jeder Sorte darfst du benutzen?

(b) Zeichne schematisch auf, wie du die Dezimalzahl 28 im Fünfersystem darstellen würdest.

(c) Welcher Dezimalzahl entspricht die Fünferzahl 123?

Aufgabe 6 – Fünferpotenzen summieren

Schaffst du es, den Wert einer im Fünfersystem geschriebenen n-stelligen Zahl $d_{n-1}d_{n-2}\ldots d_2 d_1 d_0$ mathematisch mit Hilfe des Summenzeichens $\sum$ aufzuschreiben?

[2] Ob das wirklich daher stammt, ist umstritten. Mit zwei Händen können wir ja eigentlich 11 Zustände abbilden, alles von 0 bis 10. Sollten wir also nicht im Elfersystem rechnen?

Positionale Zahlensysteme

Das Dezimalsystem ist ein Spezialfall der allgemeineren Notation eines **positionalen Zahlensystems**. Positional bedeutet hier, dass die Position der Ziffer deren Wertigkeit beeinflusst. Eine Ziffer d an der letzten Stelle ist *weniger Wert* als d an der zweitletzten Stelle.

Ein positionales System definiert eine **Basis**. Diese Basis bestimmt einerseits, **wie viele Ziffern** wir pro Stelle zur Verfügung haben. Andererseits bestimmt sie auch die **Wertigkeit** der einzelnen Stellen. Ein Zahlensystem mit Basis B hat im Allgemeinen B verschiedene Ziffern, nämlich $0, 1, 2 \ldots B-1$. Die Wertigkeit der verschiedenen Stellen hängt ebenfalls von B ab. Die letzte Stelle hat die Wertigkeit B^0, also 1, die zweitletzte B^1, gefolgt von B^2 usw.

Wenn wir also den Wert einer solchen Zahl $d_{n-1}\, d_{n-2} \ldots d_2 d_1 d_0$ bestimmen möchten, rechnen wir

$$\sum_{i=0}^{n-1} d_i \cdot B^i$$

Setzen wir statt B die Zahl 10 ein, so erhalten wir wieder die bekannte Formel für das Dezimalsystem, wenn wir $B = 5$ wählen, so entsteht das Fünfersystem.

Weitere Zahlensysteme

Neben dem bei uns dominanten Dezimalsystem finden oder fanden diverse andere Zahlensysteme Verwendung. Das **römische Zahlensystem** beispielsweise ist kein klassiches positionales System. Es besteht aus den Ziffern `I` (Wert 1), V (5), X (10), L (50), C (100), D (500) und M (1000). Eine Zahl wird ganz einfach als Summe dieser Ziffern gebildet, wobei man die grössten zu Beginn nennt. 15 wird also beispielsweise als XV geschrieben. Eine kleine Ausnahme gibt es aber: Steht eine kleinere Ziffer *vor* einer Ziffer mit grösserem Wert, so wird sie von letzterer abgezogen. Die Zahl 4 wird also als `IV` dargestellt, CD bezeichnet die Dezimalzahl 400.
Noch vor dem römisches System dominierte im mesopotamischen Reich das **babylonische System**. Dieses war eines der ersten positionalen Systeme und basierte auf der Zahl 60. Relikte davon finden wir heute noch im Alltag: Eine Stunde unterteilen wir beispielsweise in 60 Minuten.

Manche Leute plädieren gar für die Adaptierung des Zwölfersystems:

Das Binärsystem

Wenn wir Zahlen auf einem Rechner speichern möchten, können wir aus den bereits vorgestellten Gründen nicht das Dezimalsystem verwenden: Wir haben nämlich nicht genug verschiedene Ziffern zur Verfügung! Statt der benötigten 10 haben wir nur deren zwei: 0 und 1. Es bleibt uns also quasi nichts anderes übrig, als das Zweiersystem zu verwenden!

Das Binärsystem

Das **Binärsystem**, oft auch **Zweiersystem** genannt, ist das Zahlensystem mit **Basis 2**. Es verwendet also nur zwei verschiedene Ziffern (0 und 1). Die Wertigkeiten der einzelnen Stellen sind als **Potenzen der Basis 2** gegeben.
Um Verwirrung vorzubeugen, markieren wir Zahlen, die wir im Binärsystem schreiben, mit einem kleinen 2 am Schluss: 1001_2.

Bei einer Zahl im Zweiersystem stehen also ganz rechts die *Einer* (2^0), danach folgen die *Zweier* (2^1), die *Vierer* (2^2), die *Achter* (2^3), die *Sechzehner* (2^4) usw.

Beispiel – Binärzahlen lesen

Wir wollen die Zahl 1011_2 ins Zehnersystem umrechnen. 1011_2 bedeutet nichts anders als 1 Achter, 0 Vierer, 1 Zweier und 1 Einer. Wenn wir dies zusammenzählen, erhalten wir

$$1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 8 + 2 + 1 = 11$$

Im Allgemeinen können wir eine n-stellige Binärzahl wiederum als Folge von (binären) Ziffern $b_{n-1}\, b_{n-2} \ldots b_2 b_1 b_0$ bezeichnen. Den Wert der Zahl berechnen wir analog zum Dezimalsystem wie folgt:

$$b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

also einfach die Summe der Wertigkeiten multipliziert mit ihrer Ziffer (0 oder 1). Im Beispiel oben (1011_2) ist also $b_3 = 1$, $b_2 = 0$, $b_1 = 1$ und $b_0 = 1$, der Wert ist also

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 11$$

Und jenseits der Null?

Wie schreibt man eigentlich negative Zahlen im Binärsystem? Diese Frage ist leider nicht ganz so einfach zu beantworten, wie es zunächst scheint: Computer kennen ja keine Minus-Zeichen…Wir machen uns das Leben hier einiges einfacher, wenn wir uns auf positive Zahlen beschränken. Für interessierte: Das Zauberwort lautet hier *Zweierkomplement*.

Beispiel – Mehr Binärzahlen

Die Zahl 101_2 setzt sich nach obiger Logik aus einem Einer und einem Vierer zusammen. Der Wert ist also 5.

Die Zahl 10101_2 bedeutet: 1 Sechzehner, 1 Vierer, 1 Einer, also zusammen 21.

Aufgabe 7 – Binär zu Dezimal

Stelle folgende Binärzahlen im Zehnersystem dar!

(a) 10_2

(b) 110_2

(c) 111_2

(d) 1000_2

(e) 1010_2

(f) 10001_2

(g) 10001000_2

Aufgabe 8 – Grösste Binärzahl

(a) Welches ist die grösste Binärzahl, die du mit 3 Ziffern darstellen kannst? Was ist ihr Wert?

(b) Welches ist die grösste Binärzahl, die du mit 4 Ziffern darstellen kannst? Was ist ihr Wert?

(c) Welches ist die grösste Binärzahl, die du mit 5 Ziffern darstellen kannst? Was ist ihr Wert?

(d) Analysiere deine Antworten der ersten drei Teilaufgaben. Siehst du Gemeinsamkeiten? Welches ist die grösste Binärzahl, die du mit n Ziffern darstellen kannst? Was ist ihr Wert?

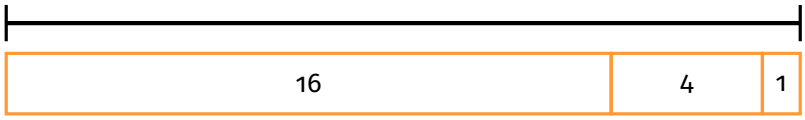
Vom Dezimal- ins Binärsystem umrechnen.

Als nächstes wollen wir uns anschauen, wie wir eine Dezimalzahl ins Zweiersystem umrechnen können. Wir schauen uns hier zwei Varianten an.

Zuerst erinnern wir uns an unsere Stäbchennotation und wollen diese für das Zweiersystem etwas abwandeln. Die Längen der Stäbchen entsprechen wiederum den Potenzen der Basis, oder anders ausgedrückt der Wertigkeit der einzelnen Stellen. Im Binärstäbchensystem haben wir also Einer-Stäbchen, Zweier-Stäbchen, Vierer-Stäbchen, Achter-Stäbchen usw. Von jeder Sorte haben wir aber nur jeweils eines zur Verfügung!

Beispiel – Binärstäbchen legen

Die Dezimalzahl 21, bzw. eine Strecke der Länge 21, können wir nach obigem Verfahren also wie folgt mit unseren Binärstäbchen legen:



Wir verwenden also ein Sechzehnerstäbchen, ein Viererstäbchen und ein Einerstäbchen. Wir müssen nun an den entsprechenden Stellen eine 1 eintragen, und an allen anderen Stellen eine 0. Oder anders ausgedrückt: Wir verwenden einen Sechzehner, null Achter, einen Vierer, null Zweier und einen Einer. Die Binärzahl 10101_2 entspricht also der Dezimalzahl 21.

Stell dir vor, wir wollen wiederum eine gewisse Strecke oder Länge mit diesen Stäbchen genau abdecken. Wenn wir die Länge der Strecke kennen (als Dezimalzahl) und diese mit unseren Binärstäbchen legen, so können wir daraus die Binärdarstellung der Zahl ablesen! Wir gehen dabei genau gleich vor wie beim Legen von Dezimalstäbchen: Wir suchen zuerst das längste Stäbchen, welches noch in die Strecke reinpasst. Für die übrig bleibende Strecke wiederholen wir das Verfahren, bis wir die Strecke genau abgedeckt haben.

Aufgabe 9 – Binärstäbchen legen

Welche Stäbchen würdest du verwenden, um eine Strecke der Länge 13 abzudecken?

Wie sieht es bei einer Strecke der Länge 42 aus? Vergleiche deine Antwort mit derjenigen aus dem Beispiel oben. Fällt dir etwas auf?

Natürlich können wir das Verfahren auch ganz ohne Stäbchen anwenden, wie das folgende Beispiel zeigt:

Beispiel – Dezimalzahlen ins Binärsystem umrechnen – Variante 1

Wir wollen die Zahl 42 als Binärzahl darstellen. Wir suchen deshalb die grösste Zweierpotenz, die noch kleiner als 42 ist, und finden $32 = 2^5$. Wir markieren an der sechsten Stelle (von rechts) also eine 1: $1\ _\ _\ _\ _\ __2$, und subtrahieren 32 von 42, erhalten also 10.

Weiter geht es also mit 10. Hier passt eine $8 = 2^3$ rein, wir notieren an der vierten Stelle von rechts also eine 1: $1\ _\ 1\ _\ _\ __2$ und rechnen $10 - 8$, machen als nächstes also mit der 2 weiter.

2 ist bereits eine Zweierpotenz (2^1), wir notieren an der zweiten Stelle von rechts also erneut eine 1: $1\ _\ 1\ _\ 1\ __2$. Wenn wir 2 von 2 abziehen, sind wir bei 0 angelangt, sind also fertig.

Alles was noch jetzt bleibt, ist, die fehlenden Stellen mit Nullen auszufüllen. Et voilà: die Dezimalzahl 42 entspricht also der Binärzahl 101010_2.

Um eine Dezimalzahl im Zweiersystem darzustellen, suchen wir also als erstes die grösste Zweierpotenz, die noch in die Zahl hineinpasst. Wir notieren an der entsprechenden Stelle eine 1 und subtrahieren diese Zweierpotenz von der Dezimalzahl. Dies wiederholen wir, bis die Dezimalzahl bei 0 angekommen ist. Stellen, die wir nicht mit einer 1 gefüllt haben, füllen wir anschliessend mit einer 0.

Aufgabe 10 – Dezimal zu binär

Stelle folgende Dezimalzahlen im Binärsystem dar, indem du das eben gelernte Verfahren anwendest.

(a) 3

(b) 6

(c) 10

(d) 11

(e) 27

(f) 40

(g) 128

Die zweite Möglichkeit, eine Dezimalzahl in die Binärdarstellung umzurechnen, basiert auf der Ganzzahldivision sowie deren Rest:

Beispiel – Dezimalzahlen ins Binärsystem umrechnen – Variante 2

Wir wollen wieder die Zahl 42 umwandeln und gehen dazu wie folgt vor:

42	:	2	=	21	Rest 0
21	:	2	=	10	Rest 1
10	:	2	=	5	Rest 0
5	:	2	=	2	Rest 1
2	:	2	=	1	Rest 0
1	:	2	=	0	Rest 1

Die Zahl können wir nun in der Rest-Spalte ablesen, und zwar von unten nach oben: 101010_2.

Dieses Verfahren können wir allgemein wie folgt beschreiben:

Umrechnung einer Dezimalzahl ins Binärsystem

Wir beginnen wieder mit der umzurechnenden Dezimalzahl und nennen diese d. Danach gehen wir wie folgt vor:
Solange d grösser als 0 ist, wiederhole folgende Schritte:

1. Berechne d mod 2 und notiere dir das Resultat [3].
2. Setze $d = d \,//\, 2$, also $d = \lfloor \frac{d}{2} \rfloor$

Die Binärdarstellung der Dezimalzahl kannst du jetzt an den Resten ablesen, und zwar in umgekehrter Reihenfolge.

Aufgabe 11 – Dezimal zu binär

Stelle folgende Dezimalzahlen im Binärsystem dar. Versuche, beide Umrechungsvarianten mindestens einmal anzuwenden.

(a) 3

(b) 6

(c) 10

(d) 11

(e) 15

(f) 27

(g) 40

(h) 128

(i) 255

[3] mod bezeichnet hier den Rest-Operator, den du in Python als % kennen gelernt hast.

Wie(so) funktioniert dieses Verfahren?

Auf den ersten Blick wirkt dieses Verfahren wie ein Zaubertrick. Es leuchtet vielleicht anfangs nicht sofort ein, wieso durch wiederholtes Halbieren und Notieren des Restes am Ende die Dezimaldarstellung herauskommt. Falls du die Magie hinter diesem 'Trick' verstehen möchtest, können dir die nachfolgenden Zeilen helfen.

Wir erinnern uns daran, dass der Wert einer Binärzahl $b_3 b_2 b_1 b_0$ als

$$b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

gegeben ist. Wenn wir wiederholt einen Faktor 2 ausklammern, können wir dies auch wie folgt schreiben:

$$2 \cdot (2 \cdot (2 \cdot b_3 + b_2) + b_1) + b_0$$

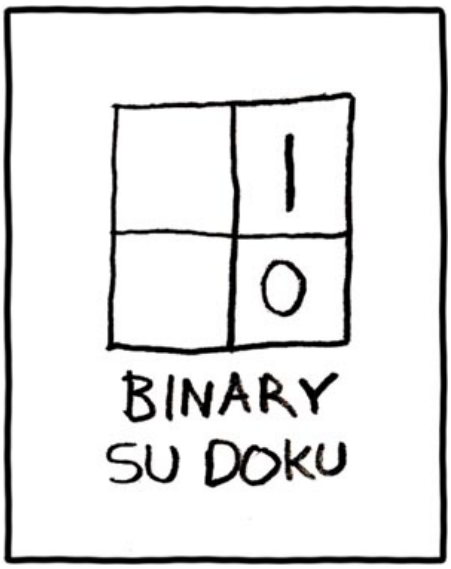
Diese Darstellung hilft uns, das obige Verfahren zu verstehen. Wenn wir hier nämlich durch 2 teilen, so erhalten wir b_0 als Rest. Und tatsächlich ist der erste Rest-Term die letzte Ziffer der Zahl! Wenn wir danach mit dem Rest des Wertes, also mit $2 \cdot (2 \cdot b_3 + b_2) + b_1$ weiterfahren und wiederum durch 2 teilen, so ergibt sich die zweitletzte Stelle, nämlich b_1. Das gleiche Vorgehen liefert uns dann auch noch b_2 und b_3, und wir können die Zahl von unten nach oben ablesen.

Aufgabe 12 (★) – Verallgemeinerung

Das zweite Verfahren kann übrigens für jedes beliebige positionale Zahlensystem angewendet werden, wenn es ein bisschen angepasst wird. Findest du heraus, wie man mit Hilfe dieser Methode eine Zahl ins Fünfersystem umrechnen kann?
Beschreibe das Vorgehen und führe damit eine Beispielumwandlung der Dezimalzahl 103 ins Fünfersystem durch.

Das Binärsystem mag jetzt auf den ersten Blick kompliziert erscheinen und ist sicherlich ein wenig gewöhnungsbedürftig. Aber es ist ein wichtiger Grundbaustein in der Informatik, auf dem wir immer und immer wieder aufbauen werden. Es lohnt sich also definitiv, sich mit dieser Repräsentation von Zahlen vertraut zu machen und sie sicher zu beherrschen.

Ausserdem: Nicht alles ist komplizierter im Zweiersystem, wie du in den folgenden Aufgaben herausfinden wirst.



This one is from the Red Belt collection, of 'medium' difficulty.

Quelle: xkcd, `https://xkcd.com/74/`

Aufgabe 13 – Gerade und ungerade Binärzahlen

Wie erkennst du bei einer binären Zahl, ob sie gerade oder ungerade ist? Welche Stelle musst du dafür anschauen, und warum?

Aufgabe 14 – Binärzahlen verdoppeln

Wie kannst du ganz einfach eine gegebene Binärzahl verdoppeln?
Tipp: Rechne die Zahl 3 ins Binärsystem um. Mach dasselbe dann mit der 6 und der 12, und schreibe die Zahlen danach untereinander in die folgenden Kästchen. Was stellst du fest? Warum ist das so?

Findest du eine Gemeinsamkeit mit der Verzehnfachung im Dezimalsystem? Welche?

Aufgabe 15 – Ver-x-fachen

Aufbauend auf deinen Erkenntnissen der letzten Aufgabe: Wie kannst du eine Binärzahl vervierfachen? Wie kannst du sie verachtfachen?

Aufgabe 16 – Binärzahlen schriftlich addieren

(a) Addiere also zuerst die beiden Dezimalzahlen 181 und 44 schriftlich.

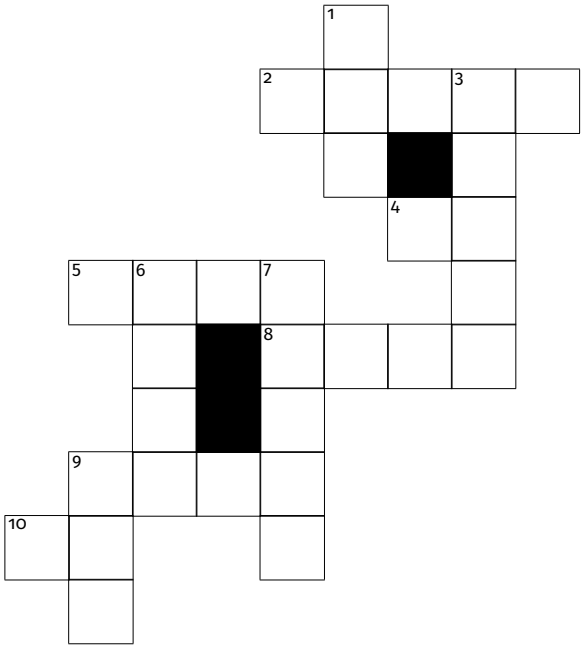
(b) Schaffst du das nun auch mit zwei Binärzahlen? Beschreibe dein Vorgehen und teste es, indem die die obigen Zahlen in ihrer Binärdarstellung (10110101_2 und 00101100_2) schriftlich zusammenzählst.

(c) Überprüfe danach, dass das Resultat stimmt, indem du es ins Zehnersystem umwandelst und mit deiner ersten Lösung aus (a) vergleichst.

(d) Welche Gemeinsamkeiten erkennst du zwischen dem Vorgehen fürs Dezimalsystem und demjenigen fürs Binärsystem? Wo liegen die Unterschiede?

Aufgabe 17 – Binäres Kreuzworträtsel

Löse das folgende binäre Kreuzworträtsel. Als Hinweise findest du unten jeweils die Zahlen im Zehnersystem, die du ins Zweiersystem umwandeln und dann ins Rätsel oben eintragen musst. Waagrechte Zahlen werden dabei von links nach rechts geschrieben (die kleinste Stelle, die Einer, stehen also ganz rechts), bei senkrechten Lösungen stehen die Einer jeweils ganz unten.



Waagrecht:

2. 18
4. 2
5. 15
8. 10
9. 8
10. 3

Senkrecht:

1. 5
3. 16
6. 12
7. 24
9. 7

Das Hexadezimalsystem

Das Binärsystem verwendet zwei verschiedene Ziffern: 0 und 1. Beim Dezimalsystem mit der Basis 10 haben wir deren 10, nämlich die Ziffern 0 - 9. Aber warum soll bei 10 Ziffern Schluss sein? Gibt es positionale Zahlensysteme, deren Basis grösser als 10 ist? Und wenn ja, was für Zeichen verwendet man dort als Ziffern, wenn doch alle unsere Zahlenziffern schon aufgebraucht sind?

Eine mögliche Antwort darauf liefert das **Hexadezimalsystem**. Dabei handelst es sich um ein in der Informatik äusserst beliebtes positionales Zahlensystem, welches als Basis die Zahl 16 verwendet. Als Ziffern benutzt man hier einfach zusätzlich zu den Zahlen 0 bis 9 die ersten paar Buchstaben des Alphabets. So steht A für 10, B für 11, C für 12, D für 13, E für 14 und F für 15.

Um zu verdeutlichen, dass eine Zahl hexadezimal notiert ist, benutzt man meist den Präfix `0x`, wie folgende Tabelle verdeutlicht:

Dez	Bin	Hex
0	`0000`	`0x0`
1	`0001`	`0x1`
2	`0010`	`0x2`
3	`0011`	`0x3`
4	`0100`	`0x4`
5	`0101`	`0x5`
6	`0110`	`0x6`
7	`0111`	`0x7`
8	`1000`	`0x8`
9	`1001`	`0x9`
10	`1010`	`0xA`
11	`1011`	`0xB`
12	`1100`	`0xC`
13	`1101`	`0xD`
14	`1110`	`0xE`
15	`1111`	`0xF`

Aufgabe 18 – Modulo 16

> Schaffst du es, das Verfahren zur Umrechnung einer Dezimalzahl in eine Binärzahl so anpassen, dass du damit eine Dezimalzahl ins Sechzehnersystem umwandeln kannst?

Beispiel – DEC to HEX

Wir wollen die Dezimalzahl 36 ins Hexadezimalsystem umrechnen. 36 besteht aus 2 Sechzehnern und 4 Einern, entspricht also der Hex-Zahl `0x24`.

Natürlich können wir hier auch jenes Verfahren etwas anpassen, welches wir zur Umwandlung ins Zweiersystem kennen gelernt haben: mit einer kleinen Änderung: Wir notieren uns jeweils den Rest modulo 16, und führen anschliessend eine Ganzzahldivision durch 16 aus. Zur Umrechnung der Dezimalzahl 123 ins Hexadezimalsystem gehen wir also wie folgt vor:

123	:	16	=	7	Rest 11
7	:	16	=	0	Rest 7

Wenn wir die 11 durch die Ziffer B ersetzen, so lesen wir von unten nach oben das Resultat ab: 7B

Aufgabe 19 – DEC to HEX

Rechne nun die folgenden Dezimalzahlen ins Hexadezimalsystem um:

(a) 4

(b) 15

(c) 16

(d) 127

(e) 255

Aufgabe 20 – Bits, Bytes und das Hexadezimalsystem

Wie du aus der obigen Tabelle ableiten kannst, können wir alle Binärzahlen mit 4 oder weniger Stellen mit einer einstelligen Hexadezimalzahl ausdrücken. Wie viele Hexadezimalstellen benötigst du, um eine Byte, also eine Binärzahl mit 8 Stellen, zu schreiben?

Gib nun ein Verfahren an, wie du Bytes schnell in hexadezimaler Form (und umgekehrt) schreiben kannst!

Beispiel – BIN to HEX

Wir wollen die Binärzahl `0110 1100` ins Hexadezimalsystem umrechnen. Das schöne am Hexadezimalsystem ist, dass wir Binärzahlen relativ einfach und vor allem kompakt umwandeln können. Pro Byte benötigen wir genau zwei Hex-Ziffern, oder anders ausgedrückt: Jede Hex-Ziffer deckt 4 Binärziffern ab. Dazu reicht es also, jeweils die einzelnen 4-Bit-Blöcke einzeln umzurechnen. `0110` entspricht der Dezimalzahl 6, welche im Hexadezimalsystem unverändert bleibt: `0x6`. `1100` wird in der Dezimaldarstellung als 12 repräsentiert, in der Hexidezimaldarstellung allerdings als `0xC`. Die gesamte Zahl drücken wir in Hex also als `0x6C` aus.

Beispiel – HEX to BIN

Wir wollen die Hex-Zahl `0xACE` ins Binärsystem übersetzen. Dafür können wir die drei Stellen separat betrachten: A entspricht der Binärzahl `1010`, C entspricht `1100` und E wird als `1110` übersetzt. Zusammengesetzt ergibt sich also `101011001110`.

Aufgabe 21 – Vom Sechzehnersystem ins Binär- und Dezimalsystem umrechnen

Rechne die folgenden Hexadezimalzahlen sowohl ins Binär- als auch ins Dezimalsystem um.

(a) `0xC`

(b) `0xBA`

(c) `0x7A`

(d) `0xABBA`

(e) `0xFACE`

Aufgabe 22 – Und jetzt alles wild gemischt

Fülle die untenstehende Tabelle aus.
Jede Zeile bschreibt die gleiche Zahl, einfach in unterschiedlichen Zahlensystemen. Der Spaltenkopf bestimmt, in welchem Zahlensystem die Zahlen in dieser Spalte geschrieben sind.
Die erste Reihe ist als Beispiel gegeben.

Dezimalsystem	Binärsystem	Hexadezimalsystem	Siebnersystem
13	`1101`	`0xD`	16
10			
	`11011`		
		`0x2F`	
			103

Zusammenfassung

Das Dezimal- oder **Zehnersystem** kanntest du bestimmt schon vor dem Lesen dieses Kapitels. Wir haben dieses hier nun etwas formaler unter die Lupe genommen, als Beispiel eines **positionalen Zahlensystems** mit Basis 10 und den Ziffern 0 - 9.

Weil moderne Computer leider nicht 10 verschiedene Ziffern unterscheiden können, sondern nur die binären Ziffern 0 und 1 kennen, können diese nicht einfach mit dem uns gängigen Zehnersystem rechnen. Stattdessen verwenden sie das **Zweiersystem** (**Binärsystem**), ein positionales Zahlensystem mit Basis 2, um ganze Zahlen zu repräsentieren.

Ein weiteres beliebtes Zahlensystem ist das **Hexadezimalsystem**, welches auf der Zahl 16 basiert. Dabei werden zusätzlich zu den Zahlen 0 - 9 die Buchstaben A - F als Ziffern verwendet.

Du hast gelernt, wie du Zahlen zwischen zwei positionalen Zahlensystem unterschiedlicher Basen umwandeln kannst, und bist nun in der Lage, mühelos Binär- und Hexidezimalzahlen zu ”entziffern”.

Text darstellen - Zeichenkodierung

Binary (He)art

Quelle: xkcd, `https://xkcd.com/99/`

Nachdem wir nun wissen, wie wir (ganze) Zahlen als Folge von Nullen und Einsen darstellen können, wollen wir in diesem Abschnitt erforschen, wie wir damit auch Text repräsentieren können.

Wir lernen dabei, auf was wir achten müssen, wenn wir Texte auf diese Weise ausdrücken möchten, und welche Lösungen sich im Laufe der Zeit dafür etabliert haben.

Bevor wir loslegen, wollen wir noch kurz definieren, was wir denn unter einem Text verstehen.

Texte und Alphabete

Unter einem **Alphabet** verstehen wir eine endliche Menge von Symbolen. Beispiele für Alphabete sind die lateinischen Grossbuchstaben {A, B, C, ..., X, Y, Z } oder das Computeralphabet, welches nur aus zwei Symbolen besteht: {0, 1 }
Ein **Text** ist nichts anderes als eine Folge von Symbolen aus einem bestimmten Alphabet. Wir benutzen der Abwechslung halber auch oft die Begriffe Botschaft oder Nachricht.

Da unsere Texte nicht nur aus Buchstaben, sondern auch Satzzeichen oder Sonderzeichen bestehen, verwenden wir in diesem Abschnitt den allgemeineren Ausdruck **Zeichen**, die all diese Kategorien umfasst.

? Zeichen abbilden

> Wie würdest du Zeichen abbilden, damit wir sie auf einem Computer speichern können?

Eine der wohl intuitivsten Ideen, um das vorliegende Problem zu lösen, ist, einfach jedem Zeichen eine Zahl zuzuweisen. Dem Buchstaben A können wir beispielsweise die Zahl 1 zuordnen, B weisen wir die Zahl 2 zu usw.

Eine solche Abbildung von Zeichen zu Zahlen und umgekehrt ist ein Beispiel einer Kodierung, oder kürzer Code.

Kodierung

Eine **Kodierung**, auch **Code** genannt, ist eine injektive Abbildung[4] zwischen zwei Alphabeten $\mathcal{A}$ und $\mathcal{B}$, die jedem Symbol aus $\mathcal{A}$ ein Symbol aus $\mathcal{B}$ zuordnet.
Handelt es sich bei $\mathcal{A}$ um Zeichen, so sprechen wir von einem **Zeichencode**.

Wenn wir also Zeichen mittels einem solchen Code in beispielsweise Zahlen umwandeln, so nennen wir diesen Vorgang **Kodieren**. Den umgekehrten Weg, also das Umwandeln von Zahlen in Zeichen, bezeichnen wir als **Dekodieren**.

Ein einzelnes kodiertes Zeichen nennen wir **Code-Wort**. Beim Dekodieren können wir die empfangene Nachricht also Code-Wort für Code-Wort durchgehen und zurückübersetzen.

Die wohl naheliegendste Art, Zeichen zu kodieren, ist durch eine simple Aufzählung: Wir nummerieren alle Zeichen einfach durch. Da wir diese Zeichencodes danach auf einem Computer speichern möchten, nutzen wir dafür das Binärsystem. Der Einfachheit halber beschränken wir uns hier auf die Grossbuchstaben des lateinischen Alphabets.

0	A	101	F	1010	K	1111	P	10100	U	11001	Z
1	B	110	G	1011	L	10000	Q	10101	V		
10	C	111	H	1100	M	10001	R	10110	W		
11	D	1000	I	1101	N	10010	S	10111	X		
100	E	1001	J	1110	O	10011	T	11000	Y		

Tabelle 1: Einfache Kodierungstabelle
Die Buchstaben (rechte Spalte) und die entsprechenden Code-Wörter (linke Spalte)

Das Code-Wort für den Buchstaben A ist hier also `0`, jenes für das Zeichen L lautet `1011`. Mit dieser Kodierungstabelle würden wir also HANS als `111 0 1101 10010` **kodieren**. Analog können wir die Zahlenfolge `1100 10100 10010 10011 100 10001` **dekodieren** und erhalten MUSTER.

[4] Eine Abbildung heisst injektiv, wenn es zu jedem Element der Zielmenge entweder genau ein oder gar kein Element aus der Ursprungsmenge gibt. Für eine injektive Funktion f gilt also: Falls $f(x) = f(y)$, so ist $x = y$.

Das funktioniert soweit ja ganz gut, wenn wir selbst für uns ein paar Zeichen kodieren. Was geschieht aber, wenn wir diese kodierten Zeichen mit anderen Personen austauschen möchten? Beispielsweise in Form einer Chat-Nachricht, oder einem Email? Dann können wir nur hoffen, dass der Empfänger der Nachricht die gleiche Kodierungstabelle wie wir verwendet, anonsten wird seine dekodierte Nachricht nicht viel Sinn ergeben! Hätten wir beispielsweise nicht bei `0`, sondern bei `1` angefangen zu zählen, würden wir die obige Zahlenfolge `1100 10100 10010 10011 100 10001` plötzlich als NVTUFS dekodieren und würden uns fragen, was der Sender denn damit meint.

Beim Austauschen von kodierten Nachrichten (beispielsweise über das Internet) ist es also wichtig, dass der Sender und der Empfänger der Nachricht den gleichen Code verwenden! Ansonsten kann es sehr schnell zu Missverständnissen kommen!

? Verwechslungsgefahr?

> Siehst du ein weiteres Problem bei der eben vorgestellten Kodierung? Kannst du dieses anhand eines Beispiels verdeutlichen?

Eindeutige Kodierungen

Ein Problem, das wir elegant umschifft haben, ist das Erkennen von einzelnen Zeichen innerhalb einer Bitfolge. In den Beispielen vorher haben wir zwischen Zeichen jeweiles einen Leerschlag gesetzt. Für uns Menschen scheint dies absolut logisch und intuitiv, Computer stellt dies allerdings vor ein Problem: diese kennen nämlich auf dieser logischen Ebene keine Leerzeichen! Alles, was sie kennen, sind Nullen und Einsen – wir müssen also ohne Trennzeichen wie etwa Leerschläge auskommen! Dass dies etwas mehr als ein simples Löschen Letzterer benötigt, soll folgendes Beispiel aufzeigen:

Beispiel – Mixed (up) signals

Alice möchte Bob gerne eine Nachricht übermitteln. Als Kodierung verwendet sie diejenige aus Tabelle 1, welche Bob ebenfalls zur Hand hat. Was Alice Bob gerne endlich mal sagen möchte, ist `ILOVEYOU`.

Sie übersetzt also alle Zeichen einzelnen und fügt diese anschliessend zu einem Bitstring – der zu übermittelnden Nachricht – zusammen. Das `I` wird zu `1000`, das `L` zu `1011` und so weiter. Schlussendlich erhält sie also die Nachricht

```
10001011111010101100110001111010100
```

und übermittelt diese an Bob. Dieser freut sich extrem, eine Nachricht von Alice bekommen zu haben. Nur hat er seine liebe Mühe, die Botschaft zu entschlüsseln. Schon beim ersten Buchstaben gerät er ins Stocken…wo endet denn dieser? Handelt es sich um ein B, gefolgt von drei A? Oder ein C gefolgt von zwei A? Oder ein E gefolgt von nur einem A? Oder doch ein I? Oder etwa ein R? Alle diese Varianten wären möglich, je nach dem wo man sich die Leerschläge ”denkt”:

```
1 0 0 0 1...
10 0 0 1...
100 0 1...
1000 1...
10001...
```

Dass er die richtige Kombination von Buchstaben trifft, ist extrem unwahrscheinlich – es gibt einfach zu viele davon! Eine Mögliche wäre beispielsweise

```
10001 0 1111 101 0 101 100 11000 111 0 10 100
```

und das bedeutet so viel wie RAPFAFEYHACE. Bob wird daraus allerdings nicht schlau. Da er nicht alle Möglichkeiten durchprobieren möchte, gibt er enttäuscht auf.

Wie das letzte Beispiel zeigt, ist eine Nachricht, die mittels dem Zeichencode aus Tabelle 1 kodiert wurde, **nicht eindeutig dekodierbar**. Eine Nachricht kann also vom Empfänger anders interpretiert werden, als dies vom Sender gedacht war!

Zeichencodes, bei denen dieses Problem nicht auftritt, nennen wir demzufolge **eindeutig dekodierbar**, und wollen uns als Nächstes anschauen, wie wir unseren naiven Zeichencode in einen solchen umwandeln können.

? Zeichencodes eindeutig machen

> Was für Möglichkeiten siehst du, den Zeichencode aus Tabelle 1 eindeutig zu machen?

Eine Art von Zeichencodes, die zu den eindeutig dekodierbaren gehören, sind die so genannten **Zeichencodes mit fixer Länge** (engl. fixed length encodings). Bei diesen hat jedes Code-Wort eine fixe Länge, und diese Länge ist für alle Code-Wörter gleich. Das Dekodieren ist danach ganz einfach. Wenn wir wissen, wie lange die einzelnen Code-Wörter sind, so können wir diese ganz einfach identifizieren und übersetzen.

Wir wollen unseren naiven Zeichencode aus Tabelle 1 in einen solchen Zeichencode fixer Länge umwandeln. Dafür füllen wir alle Zahlen einfach von links mit Nullen auf, bis alle Code-Wörter dieselbe Länge haben. Dies resultiert in folgender Tabelle:

00000	A	00101	F	01010	K	01111	P	10100	U	11001	Z
00001	B	00110	G	01011	L	10000	Q	10101	V		
00010	C	00111	H	01100	M	10001	R	10110	W		
00011	D	01000	I	01101	N	10010	S	10111	X		
00100	E	01001	J	01110	O	10011	T	11000	Y		

Tabelle 2: Eindeutige Kodierung mit fixer Länge
Die Buchstaben (rechte Spalte) und die entsprechenden Code-Wörter (linke Spalte)

Mit diesem Zeichencode wird also HANS als `00111000000110110010` kodiert - die vier Zeichen werden je einem Code-Wort mit einer Länge von 5 Bits zugeordnet, die gesamte Nachricht ist also 20 Bits lang.

Nun können wir auch das zweite Beispiel eindeutig dekodieren! Bei der Nachricht `01100101001001010011001001000 1` wissen wir, dass jedes Zeichen jeweils mit 5 Bits kodiert wurde und können diese also eindeutig unterscheiden:
`01100 10100 10010 10011 00100 10001` wird folglich zu MUSTER.

Aufgabe 23 – Liebesbotschafter

> Kannst du Alice helfen, ihre Botschaft aus obigem Beispiel (ILOVEYOU) mittels dem Zeichcode aus Tabelle 2 so zu kodieren, dass Bob die Nachricht eindeutig dekodieren kann?

Aufgabe 24 – ÜbersetzerIn spielen

(a) Kodiere die Worte KLOSTER und SCHULE mit dem Zeichencode aus Tabelle 2

(b) Dekodiere die Nachricht `100101001101000010110011`

(c) Dekodiere folgende Botschaft:
`000010100001101000001000111000100010111000010010101010`

Aufgabe 25 – Fehlende Zeichen

Wie du vielleicht gemerkt hast, werden in Tabelle 2 noch nicht alle fünfstelligen Code-Wörter aktiv genutzt. Wie viele davon sind noch ungenutzt? Welche sind das?

Welche Zeichen oder Buchstaben würdest du dem Zeichencode hinzufügen, also den übrig gebliegeben Code-Wörtern zuordnen?

Ein erster Standard: ASCII

In den 1960er-Jahren hat sich – zumindest im englischsprachigen Raum – ein erster Standard für eine Kodierungstabelle wie Tabelle 2 etabliert: **ASCII**. ASCII steht für American Standard Code for Information Interchange und gehört zur Familie der Kodierungen mit fixer Länge. Er verwendet für jedes Code-Wort 7 Bits, kann also gesamthaft 128 verschiedene Code-Wörter unterscheiden.
Die mit ASCII kodierbare Zeichen umfassen das lateinische Alphabet (sowohl Gross- als auch Kleinbuchstaben), die zehn uns bekannten Ziffern (0-9), einige Sonder- und Satzzeichen (Leerschlag, Punkt, Komma, $, %, …) sowie ein paar so genannte nicht druckbare Zeichen. Man kann sich die Informationsübertragung von anno dazumals wirklich so vorstellen, als würde der Empfänger an einer mechanischen Schreibmaschine sitzen und den empfangenen Text sowie die Befehle direkt abtippen oder ausführen. Ein paar Beispiele von nicht druckbaren Zeichen sind etwa DEL (*delete*), um ein Zeichen zu löschen oder LF (*line feed*), um auf die nächste Zeile zu springen. Bei einem NUL endet die Übertragung, die Nachricht ist zu Ende.

Dez	Hex	Bin	Zeichen	Dez	Hex	Bin	Zeichen
0	0x00	0000000	NUL	32	0x20	0100000	SPACE
1	0x01	0000001	SOH	33	0x21	0100001	!
2	0x02	0000010	STX	34	0x22	0100010	“
3	0x03	0000011	ETX	35	0x23	0100011	#
4	0x04	0000100	EOT	36	0x24	0100100	$
5	0x05	0000101	ENQ	37	0x25	0100101	%
6	0x06	0000110	ACK	38	0x26	0100110	&
7	0x07	0000111	BEL	39	0x27	0100111	’
8	0x08	0001000	BS	40	0x28	0101000	(
9	0x09	0001001	TAB	41	0x29	0101001	)
10	0x0A	0001010	LF	42	0x2A	0101010	*
11	0x0B	0001011	VT	43	0x2B	0101011	+
12	0x0C	0001100	FF	44	0x2C	0101100	,
13	0x0D	0001101	CR	45	0x2D	0101101	-
14	0x0E	0001110	SO	46	0x2E	0101110	.
15	0x0F	0001111	SI	47	0x2F	0101111	/
16	0x10	0010000	DLE	48	0x30	0110000	0
17	0x11	0010001	DC1	49	0x31	0110001	1
18	0x12	0010010	DC2	50	0x32	0110010	2
19	0x13	0010011	DC3	51	0x33	0110011	3
20	0x14	0010100	DC4	52	0x34	0110100	4
21	0x15	0010101	NAK	53	0x35	0110101	5
22	0x16	0010110	SYN	54	0x36	0110110	6
23	0x17	0010111	ETB	55	0x37	0110111	7
24	0x18	0011000	CAN	56	0x38	0111000	8
25	0x19	0011001	EM	57	0x39	0111001	9
26	0x1A	0011010	SUB	58	0x3A	0111010	:
27	0x1B	0011011	ESC	59	0x3B	0111011	;
28	0x1C	0011100	FS	60	0x3C	0111100	<
29	0x1D	0011101	GS	61	0x3D	0111101	=
30	0x1E	0011110	RS	62	0x3E	0111110	>
31	0x1F	0011111	US	63	0x3F	0111111	?

Dez	Hex	Bin	Zeichen	Dez	Hex	Bin	Zeichen
64	0x40	1000000	@	96	0x60	1100000	‘
65	0x41	1000001	A	97	0x61	1100001	a
66	0x42	1000010	B	98	0x62	1100010	b
67	0x43	1000011	C	99	0x63	1100011	c
68	0x44	1000100	D	100	0x64	1100100	d
69	0x45	1000101	E	101	0x65	1100101	e
70	0x46	1000110	F	102	0x66	1100110	f
71	0x47	1000111	G	103	0x67	1100111	g
72	0x48	1001000	H	104	0x68	1101000	h
73	0x49	1001001	I	105	0x69	1101001	i
74	0x4A	1001010	J	106	0x6A	1101010	j
75	0x4B	1001011	K	107	0x6B	1101011	k
76	0x4C	1001100	L	108	0x6C	1101100	l
77	0x4D	1001101	M	109	0x6D	1101101	m
78	0x4E	1001110	N	110	0x6E	1101110	n
79	0x4F	1001111	O	111	0x6F	1101111	o
80	0x50	1010000	P	112	0x70	1110000	p
81	0x51	1010001	Q	113	0x71	1110001	q
82	0x52	1010010	R	114	0x72	1110010	r
83	0x53	1010011	S	115	0x73	1110011	s
84	0x54	1010100	T	116	0x74	1110100	t
85	0x55	1010101	U	117	0x75	1110101	u
86	0x56	1010110	V	118	0x76	1110110	v
87	0x57	1010111	W	119	0x77	1110111	w
88	0x58	1011000	X	120	0x78	1111000	x
89	0x59	1011001	Y	121	0x79	1111001	y
90	0x5A	1011010	Z	122	0x7A	1111010	z
91	0x5B	1011011	[	123	0x7B	1111011	{
92	0x5C	1011100	\	124	0x7C	1111100	|
93	0x5D	1011101	]	125	0x7D	1111101	}
94	0x5E	1011110	^	126	0x7E	1111110	-
95	0x5F	1011111	_	127	0x7F	1111111	DEL

Die meisten für uns interessanten Zeichen beginnen beim Code-Wort 1000001, oder dezimal ausgedrückt 65. Dort beginnt nämlich das lateinische Alphabet in Grossbuchstaben. Die entsprechenden Kleinbuchstaben starten dann bei 11000001 (97). Du siehst, die Gross- und Kleinbuchstaben unterscheiden sich in ihrer Binärdarstellung in nur einem einzigen Bit, nämlich dem zweiten von links. Eine sehr elegante Designentscheidung!

Beispiel – Deutsch – ASCII

Wir wollen das Wort `Binary` mit ASCII kodieren. Dazu verwenden wir die Tabelle oben und gehen Buchstabe für Buchstabe durch. B wird zu `1000010`, `i` zu `1101001`, das n zu `1101110` und so weiter. Schlussendlich erhalten wir
`1000010 1101001 1101110 1100001 1110010 1111001`,
wobei wir die einzelnen Buchstaben durch Leerschläge getrennt haben. Dies machen wir nur der besseren Lesbarkeit wegen, ein Computer würde die Leerschläge natürlich weglassen und stattdessen
`100001011010011101110110000111100101111001` schreiben!

Beispiel – ASCII – Deutsch

Wir erhalten die Nachricht
`101011111010001100001111010001001111110011010000011101011110000011111`,
welche mit ASCII kodiert wurde. Um diese zu entschlüsseln, betrachten wir also immer 7 Bits als eine Einheit und übersetzen diese. Das erste Code-Wort ist also `1010111` und entspricht einem W. Danach folgt `1101000`, was wir zu h übersetzen. Wenn wir so weiter vorgehen, erhalten wir die Nachricht `What's up?`.

Du siehst also, dass wir mit ASCII mehr als nur Buchstaben kodieren und dekodieren können!

Aufgabe 26 – ASCII kodieren

(a) Kodiere das Wort `Kloster` mit ASCII.

(b) Kodiere deine Schulmailadresse mit ASCII.

Aufgabe 27 – Nochmals Amor spielen

Alice versucht nun wiederum, Bob eine Liebesbotschaft zu übermitteln. Diesmal würde sie gerne ASCII verwenden, um die Nachricht zu kodieren. Kannst du ihr helfen, die Nachricht `I love you!` zu kodieren?

Aufgabe 28 – ASCII dekodieren

Du bekommst folgende Botschaft, welche mit ASCII kodiert wurde. Der Einfachheit halber sind hier alle Code-Wörter durch einen Leerschlag getrennt. Kannst du die Nachricht dekodieren?

```
1010100 1101000 1100101 0100000 1110000 1110010 1101111 1100010
1101100 1100101 1101101 0100000 1110111 1101001 1110100 1101000
0100000 1000001 1010011 1000011 1001001 1001001 0100000 1101100
1101001 1100101 1110011 0100000 1101001 1101110 0100000 1101001
1110100 1110011 0100000 1100010 1100101 1100111 1101001 1101110
1101110 1101001 1101110 1100111 0101110
```

Aufgabe 29 – Das Problem mit ASCII

Welche Probleme birgt die Zeichenkodierung mit ASCII? Gibt es Wörter, die sich damit nicht kodieren lassen? Wenn ja: welche, und weshalb?

Das Unicode-'Wunder'

Vielleicht hast du beim Betrachten der Kodiertabelle bereits einen Schwachpunkt von ASCII erkannt. Einige Zeichen, die wir in der deutschen, französischen oder anderen Sprachen nutzen, fehlen dort nämlich. ASCII kennt keine Umlaute, keine Akzente, kein Euro-Zeichen[5] und schon gar keine Emojis.

Auf die Letzteren könnten wir ja notfalls verzichten, zumal diese zur Geburtszeit von ASCII noch überhaupt nicht bekannt waren. Aber die anderen, wie auch viele andere nicht in ASCII enthaltene Zeichen, wurden von diversen Sprachen schmerzlichst vermisst. Deshalb haben diverse Länder oder Sprachgemeinschaften den ASCII-Code so erweitert, dass dieser ihren Bedarf abdeckt. Natürlich wurde dabei nicht darauf geachtet, dass die Erweiterungen eindeutig und standardisiert waren. Und genau dieser Punkt führte zu einer grossen Inkompatibilität, als das aufkommende Internet es vielen Ländern ermöglichte, über die ganze Welt hinweg digital zu kommunizieren! Ohne standardisierte Kodiertabelle ist eine solche Kommunikation, wie wir in der Einleitung zu ASCII gesehen haben, schlicht unmöglich oder zumindest sehr problematisch.

Was dann geschah, grenzt beinahe an ein Wunder. Im Jahr 1991 veröffentliche das Unicode-Konsortium – eine gemeinnützige Organisation, der unteren anderen Grössen wie Google, Adobe, Apple oder Microsoft angehören – einen neuen Standard: **Unicode**. Dieser Standard hat zum Ziel, alle verwendeten Zeichen auf der Welt aufzulisten und ihnen einen eindeutigen Schlüssel, einen so genannten **Code Point** zuzuweisen.

Bei diesen Code Points handelt es sich um nichts anderes als eine ganze Zahl, die meist in hexadezimaler Form dargestellt wird. Man könnte Code Points deshalb auch Zahlenwerte nennen. So hat der deutsche Umlaut *ä* etwa den Code Point `0xE4`[6], oder dezimal 228. Das in der Einleitung vorgestellte Schriftzeichen 木 hat den Code Point `0x6728` (dezimal 26408).

Die ersten 128 Code Points entsprechen genau den 128 Zeichen, die der ASCII Standard definiert. Dieser cleverer Designentscheid von Unicode ermöglicht eine gewisse Rückwärtskompatibilität, wie wir später sehen werden. Darüber hinaus beinhaltet Unicode die Zeichen aller bekannten Alphabete (Arabisch, Griechisch, Hebräisch, Lateinisch, …), chinesische, japanische, koreanische und andere asiatische Schriftzeichen, Währungssymbole, alle Symbole der Brailleschrift, und seit Version 6.0 auch Emoji. In der aktuellen Version (14.0) beinhaltet der Unicode-Standard 144,697 **Zeichen**!

Eine Übersicht aller Zeichen der aktuellen Unicode-Version findest du auf der offiziellen Homepage von Unicode unter diesem Link.

[5] Wie auch, die europäische Währung ist einiges jünger als der ASCII Standard!
[6] Oft sieht man auch die Notation U+E4 für Code Points

Did you know...? – Emojis

Auch die heutzutage äusserst beliebten Emojis werden vom Unicode-Konsortium verwaltet und standardisiert. Sprich: Jedes Emoji hat ebenfalls einen eigenen Code Point. Wusstest du, dass…

- …jedes Emoji eine englische Kurzbeschreibung hat? 😃 heisst offiziell *grinning face with big eyes*, 😱 wird als *face screaming in fear* beschrieben, und 🙈 ist der *see-no-evil monkey*.
- …Emojis je nach Art des Geräts (bzw. dessen Hersteller), auf welchem sie angezeigt werden, unterschiedlich aussehen?
- …😂 das am häufigsten verwendete Emoji ist? Die komplette Rangliste findest du hier.
- …Jede und Jeder einen Vorschlag für ein neues Emoji machen kann? Die Anleitung dafür findest du hier.
- …mit Unicode Version 13 endlich ein 🫕 Fondue-Emoji hinzugefügt wurde? Eine Liste mit allen neu dazu gekommenen Emojis findest du hier.
- …Emojis ihren Ursprung in Japan haben? Die Geschichte der Emojis und wie sich diese so schnell viral verbreitet haben, wird in diesem kurzen Video erklärt:

UTF-8 und UTF-32

Unicode selber spezifiziert nicht, wie diese Code Points kodiert werden sollen. Unicode ist also **keine Kodierung**, sondern lediglich ein Standard, eine Art kanonische Auflistung aller bekannten Schriftzeichen. Im Folgenden wollen wir zwei Kodierungsmethoden für diese Code Points kurz vorstellen. Es handelt sich dabei um Varianten des Unicode Transformation Formats (UTF).

Bei **UTF-32** handelt es sich um eine Kodierung fixer Länge: Jedes Unicode-Zeichen wird als 32 Bit lange Binärzahl kodiert. Diese Binärzahl entspricht genau dem Code Point von Unicode.

Beispiel – UTF-32

Der Umlaut *ä* hat den Unicode Code Point U+E4. Dies entspricht also der Hexadezimalzahl `0xE4`. Binär dargestellt erhalten wir $\texttt{1110 0100}_2$, und diese Zahl müssen wir nun noch auf 32 Bits aufstocken, indem wir ein paar Nullen vorne anhängen. Daraus ergibt sich dann folgende Kodierung nach UTF-32:

```
0000 0000 0000 0000 0000 0000 1110 0100
```

(Die eingefügten Leerzeichen dienen nur der besseren Lesbarkeit)

Aufgabe 30 – UTF-32

(a) Das Euro-Zeichen (€) hat den Code Point U+20AC. Wie sieht die UTF-32 Kodierung davon aus?

(b) Das Zeichen *å* wird mit UTF-32 als
`0000 0000 0000 0000 0000 0000 1110 0101`
kodiert. Was ist der zugehörige Code Point?

Mit UTF-32 können wir theoretisch $2^{32} = 4'294'967'296$ verschiedene Zeichen kodieren – deutlich mehr als genug für den gesamten Unicode-Standard! Ein grosser Nachteil sind allerdings die vielen führenden Nullen, diese führen zu einem grossen Speicherplatzverbrauch! Zur Erinnerung: ein ASCII-Zeichen wird mit 7 Bits kodiert. Mit UTF-32 benötigen wir dafür mehr als das Vierfache an Bits!

UTF-8 versucht dieses Problem zu addressieren. Bei UTF-8 handelt es sich nicht um eine Kodierung fixer Länge, sondern um eine **Kodierung variabler Länge**. Jedes Zeichen wird durch ein vielfaches von 8 Bit (1 Byte) kodiert. Gewisse Zeichen benötigen nur ein Byte, andere deren 2, wiederum andere 3, usw. Wir betrachten hier nur das grobe Kodierungsschema, und glauben der Einfachheit halber einfach, dass diese Kodierung eindeutig ist, ohne dies hier zu beweisen. Wir werden uns nämlich im nächsten Kapitel vertieft mit solchen Kodierungen variabler Länge auseinandersetzen, und du wirst danach ganz einfach begründen können, dass es bei dieser Kodierung trotz unterschiedlich langer Code-Wörter nicht zu Verwechslungen kommen kann!

Als erstes wird der Unicode Code Point als Binärzahl ausgedrückt. Je nach Länge dieser Zahl gilt dann eine der folgenden Regeln:

1. Alle ASCII-Zeichen, sprich die Code Points von 0 bis U+7F (127), werden eins zu eins mit einer führenden Null kodiert. Der Grossbuchstabe *A* wird in ASCII mit 01000001_2 kodiert, in UTF-8 hängen wir einfach noch eine Null vorne dran, um auf unsere 8 Bits zu kommen. Dies hat den schönen Vorteil, dass UTF-8 rückwärtskompatibel zu ASCII ist!

2. Die Code Points U+80 bis U+07FF, also alle Code Points, deren Binärdarstellung zwischen 8 und 11 Bits lang ist, werden nach dem Muster **110**xxxxx **10**xxxxxx kodiert, wobei die x durch die entsprechenden Bits ersetzt werden. Der Buchstabe ö beispielsweise hat den Code Point U+F6, oder binär ausgedrückt 1111 0110. Wenn wir diese Bits in obiges Muster rechtsbündig einsetzen, und die freien x links mit Nullen auffüllen, erhalten wir **110**00011 **10**110110

3. Code Points, deren Binärdarstellung zwischen 12 und 16 Bits benötigt, werden nach dem Muster **1110**xxxx **10**xxxxxx **10**xxxxxx kodiert. Das Zeichen ₩ hat den Code Point U+20A9), binär geschrieben 0010 0000 1010 1001. Nach Ersetzen der x im obigen Muster erhalten wir **1110**0010 **10**000010 **10**101001 als Kodierung in UTF-8.

4. Code Points mit einer längeren Binärdarstellung (bis 21 Bit) werden nach dem Muster **11110**xxx **10**xxxxxx **10**xxxxxx **10**xxxxxx kodiert.

Beispiel – UTF-8

Wir betrachten nochmals den Umlaut *ä* Der zugehörige Code Point wird binär als 1110 0100_2 geschrieben. Aufgrund der Länge dieser Zahl (8 Bits) wenden wir die zweite Regel an und ersetzen die x in **110**xxxxx **10**xxxxxx mit den entsprechenden Bits. Die UTF-8 Kodierung von *ä* lautet also

11000011 **10**1100100

wobei die kodierte Binärzahl unterstrichen hervorgehoben ist.

Aufgabe 31 – UTF-8

(a) Das Euro-Zeichen (€) hat den Code Point U+20AC. Wie sieht die UTF-8 Kodierung davon aus?

(b) Das Zeichen *ã* wird mit UTF-8 als
11000011 10100011
kodiert. Was ist der zugehörige Code Point?

Zusammenfassung

In diesem Kapitel hast du gelernt, wie Text, beziehungsweise die einzelnen Zeichen eines Textes, binär auf einem Computer repräsentiert werden. Eine solche Abbildung von Zeichen auf eine Folge von anderen Symbolen, den so genannten Code-Wörtern, nennt man **Kodierung** oder Code. Eine Kodierung heisst **eindeutig dekodierbar** oder einfach nur eindeutig, wenn eine mit ihr kodierte Nachricht eindeutig zurück übersetzt werden kann. Kodierungen fixer Länge sind eine Art solcher eindeutigen Codes, bei welchen alle Code-Wörter die gleiche Länge haben.

Damit zwei Parteien miteinander kommunizieren können, ist es unter anderem wichtig, dass beide Seiten die gleiche Kodierung verwenden. Aus diesem Grunde wurden verschiedene Standards entwickelt. Zwei Beispiele davon haben wir in diesem Kapitel kennen gelernt: die Kodierung **ASCII** und die universelle kanonische Zeichensammlung **Unicode**. Letztere ist keine Kodierung an und für sich, sondern listet lediglich alle bekannten Schriftzeichen der Welt auf. Kodiert werden diese mit Varianten des **Unicode Transformation Formats**, beispielsweise mit UTF-8 oder UTF-32.

Textkomprimierung mit Kodierungen variabler Länge

Vielleicht bist du dem **Morsecode** schon einmal begegnet. Es handelt sich dabei um eine Kodierung, bei welcher Zeichen also Folge von zwei verschiedenen Signallängen (kurz und lang) kodiert werden. Dazu kann beispielsweise ein Tonsignal verwendet werden, aber auch Lichtsignale oder Rauchzeichen sind denkbar, solange man damit zwei verschiedene Signallängen unterscheiden kann: kurz (auch *dit* genannt) und lang (*dah*). Geschrieben wird dies meist mit einem Punkt für ein dit und einem Strich für ein dah.

In Tabelle 5 findest du einen Ausschnitt aus dieser Kodierung. Jedes Zeichen wird also als Folge von dits oder dahs geschrieben, das Ende eines Buchstabens wird mit einer kurzen Pause angedeutet, das Ende eines Wortes mit einer etwas längeren Pause.

Beispiel – Save Our Souls

Ein sehr bekanntest Beispiel einer mit Morsecode kodierten Nachricht ist das Notsignal SOS, kurz für *save our souls*. Das S wird als drei dits oder Punkte kodiert, O hingegen mit drei dahs oder Strichen. Dieser Hilferuf wird lautet also

··· ––– ···

A	·–	N	–·	0	–––––
B	–···	O	–––	1	·––––
C	–·–·	P	·––·	2	··–––
D	–··	Q	––·–	3	···––
E	·	R	·–·	4	····–
F	··–·	S	···	5	·····
G	––·	T	–	6	–····
H	····	U	··–	7	––···
I	··	V	···–	8	–––··
J	·–––	W	·––	9	––––·
K	–·–	X	–··–	.	·····
L	·–··	Y	–·––	!	––··––
M	––	Z	––··	?	··––··

Tabelle 5: Ein Ausschnitt aus dem Morse-Alphabet

? Auf die Länge kommt es an

> Dir ist wahrscheinlich aufgefallen, dass gewisse Buchstaben ein viel kürzeres Code-Wort zugewiesen bekommen haben als andere. An was mag das wohl liegen? Und was für ein Vorteil ergibt sich daraus?

Du fragst dich jetzt vielleicht, weshalb wir uns im heutigen digitalen Zeitalter mit einer Kodierung herumschlagen sollen, die vor über 150 Jahren für das Versenden von Telegrammen entwickelt wurde. Die Antwort ist ganz einfach: Wir interessieren uns hier nicht für die Technik des Übermitteln, sondern für Eigenheiten dieses Kodierung!

Eine für uns interessante Eigenschaft an der Morse-Kodierung ist die Tatsache, dass nicht alle Code-Wörter gleich lang sind: es handelt sich dabei also nicht um eine Kodierung fixer Länge. Und die Code-Wörter wurden natürlich nicht zufällig so gewählt: Zeichen, die in der englischen Sprache häufiger benutzt werden, erhielten ein kürzeres Code-Wort. So haben die am häufigsten verwendeten Buchstaben E und T die kürzesten Code-Wörter: sie bestehen aus je genau einem Symbol. Selten benötigten Buchstaben wie beispielsweise dem X hingegen wurden längere Code-Wörter zugewiesen. Dass damit bei der Übertragung von Nachrichten viel Zeit gespart werden konnte, ist offensichtlich.

Gerne würden wir diesen Ansatz auch für unsere Zeichenkodierung mit dem Computeralphabet verfolgen, damit liesse sich sicherlich viel Speicherplatz und Übertragungszeit sparen! Ein erster Versuch liegt auf der Hand: Können wir nicht einfach alle Punkte durch Nullen und alle Striche durch Einsen (oder umgekehrt) ersetzen?

? Morsen mit Nullen und Einsen

> Wieso funktioniert dieser einfache Ansatz nicht? Begründe deine Antwort und gib ein problematisches Beispiel an.

Ganz so einfach ist es leider nicht. Die Begründung kennst du bereits aus einem früheren Kapitel: Diese Kodierung ist nicht eindeutig! Die Nachricht 000_2 beispielsweise können wir als S, EI, IE oder EEE auffassen. *Aber warum funktioniert das bei Morsecode?*, magst du dich jetzt fragen. Das Morse-Alphabet besteht im Gegensatz zum Computeralphabet nicht nur aus 2, sondern aus 3 Symbolen. Neben Punkten und Strichen werden nämlich auch Pausen verwendet, um das Ende von Buchstaben oder Wörtern zu signalisieren. Ein solches Trennzeichen steht uns bei der binären Kodierung leider nicht zur Verfügung, wir müssen uns mit Nullen und Einsen begnügen.

So schnell wollen wir aber nicht aufgeben! Können wir unsere binäre Morsekodierung so umbiegen, dass Code-Wörter nicht verwechselt werden können, ohne dabei aber auf eine Kodierung fixer Länge auszuweichen? Um diese Frage zu beantworten betrachten wir zunächst eine etwas andere Darstellung der klassischen Morsekodierung: Die Darstellung als Code-Baum. Diese soll uns helfen, das Problem der Eindeutigkeit aus einem anderen Blickwinkel zu betrachten:

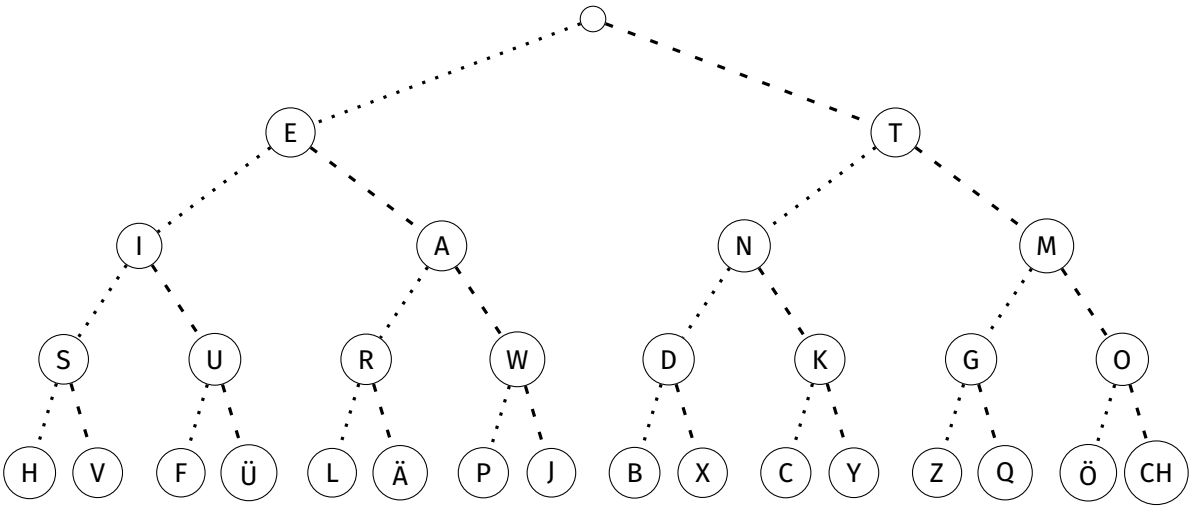


Abbildung 1: Ein Auszug der Morsekodierung als Codebaum

(Code)-Baum

Ein Codebaum ist nichts anderes als eine graphische Darstellung einer Kodierung - als Baum. Unter einem **Baum** versteht man in der Informatik etwas leicht Anderes als beispielsweise in der Botanik. Wir verzichten hier auf eine graphentheoretische Definition des Begriffs und beschränken uns lieber auf eine einfache und intuitiv nachvollziehbare Beschreibung.
Ein Baum besteht aus **Knoten**, welche durch **Kanten** (Äste) verbunden sind. Erstere sind im Beispiel des Codebaums des Morsealphabets als Kreise dargestellt. Der oberste Knoten heisst **Wurzel**. Knoten, die direkt verbunden unterhalb eines anderen Knotens liegen, wollen wir hier **Kinder** nennen. Und Knoten, die keine Kinder haben, heissen auch **Blätter**. Du siehst also: Bäume wachsen in der Informatik von oben nach unten!

Bei einem Codebaum schreiben wir unsere Codewörter in Knoten des Baumes, und verwenden die Kanten um zu verdeutlichen, mit welchen Symbolen man vom aktuellen zum nächsten Code-Wort kommt. Im Beispiel des Morse-Baums steht eine Abzweigung nach links für das Anhängen eines Punktes, während ein Abstecher nach rechts das Anhängen eines Strichs bedeutet.

Beispiel – Der Morsebaum

Wir betrachten den Codebaum für die Morsekodierung. Um das Zeichen R zu kodieren, müssen wir von der Wurzel aus kommend zuerst nach links, danach nach rechts, und schlussendlich wieder nach links abbiegen. Diese Verzweigungen stehen für Punkt, Strich und Punkt, R wird also als ·–· kodiert.

Um das Zeichen –·· zu dekodieren, müssen wir von der Wurzel aus zuerst rechts, und dann zweimal links abbiegen. Wir landen also beim Buchstaben D.

? Fehlende Trennzeichen

Kannst du mit Hilfe des Morsebaums erklären, warum diese Kodierung ohne Trennzeichen nicht eindeutig ist? Erläutere deine Überlegungen mit einem Beispiel.

Anhand dieser Darstellung als Codebaum wird uns auch schnell bewusst, weshalb unser erster Entwurf der binären Morsekodierung nicht eindeutig ist. Betrachten wir als Beispiel dafür die Symbolfolge ·–·· . Um diese zu dekodieren, starten wir wiederum bei der Wurzel des Codebaums und arbeiten uns in die Tiefe vor, indem wir entsprechend abbiegen. Zuerst links, dann rechts, und dann zweimal links. Wir landen dabei also bei L. Doch diese Folge können wir genau so gut als ·– ·· lesen, was in AI resultiert. Ohne Trennzeichen wissen wir auf unserem Weg im Codebaum nicht, wann wir beim gewünschten Code-Wort angekommen sind und wieder von vorne bei der Wurzel beginnen müssen! Und genau diese Unsicherheit ist es, welche die Kodierung in gewissen Fällen nicht eindeutig macht. Wenn wir beim Dekodieren bei W (·––) angekommen sind, sind wir auf unserem Weg an den Code-Wörtern E (·) und A (·–) vorbeigekommen. Wir hätten genau so gut dort stoppen und von vorne beginnen können! Problematisch wird es also anscheinend dann, wenn wir auf unserem Dekodierweg im Baum andere Code-Wörter passieren. Und wann ist das der Fall? Offensichtlich genau dann, wenn ein Code-Wort den Anfang eines anderen Code-Worts bildet. Oder etwas formaler ausgedrückt: Wenn ein Code-Wort ein **Präfix** eines anderen ist. Wenn wir L (·–··) anschauen, so sind die Code-Wörter E (·), A (·–) und W (·––) allesamt Präfixe dieser Symbolfolge.

Präfix und Suffix

Ein Code-Wort p ist ein **Präfix** eines anderen Code-Wortes w, wenn p den Anfang von w bildet.
Umgekehrt nennen wir ein Code-Wort s **Suffix** eines anderen Code-Wortes w, wenn w auf s endet.

Beispiel – Präfixe und Suffixe

Den Begriff Präfix kennst du vielleicht auch aus dem Gebrauch in der Linguistik der deutschen Sprache: dort nennt man es oft *Vorsilbe*. *ab* ist die Vorsilbe im Verb *ab*stimmen. Wir sagen hier: *ab* ist ein Präfix von *ab*stimmen, da das Wort *ab* den Anfang von *ab*stimmen bildet.

In der Morsekodierung bilden sowohl · (E) als auch ·– (A) den Anfang von ·–· (R), also sind beide Präfixe davon.

Ebenso sind sowohl · (E) als auch –· (N) Suffixe von ·–· (R).

Aufgabe 32 – Präfixe

Wie viele Präfixe hat das Code-Wort –··– ? Liste alle auf. Welchen Zeichen entsprechen diese?

Aufgabe 33 – Präfixe bei Kodierungen fixer Länge

Bisher haben wir hauptsächlich Kodierungen Fixer Länge betrachtet. Als Erinnerung: Bei einer solchen Kodierung sind alle Code-Wörter gleich lang. Kann hier ein Code-Wort ein Präfix eines anderen sein? Begründe deine Antwort!

Präfixfreie Kodierungen

Wie sieht eigentlich ein Codebaum für eine Kodierung fixer Länge aus? Als Beispiel betrachten wir eine solche Kodierung über dem Computeralphabet für die ersten acht Buchstaben des lateinischen Alphabets: A kodieren wir als `000`, B als `001` usw..

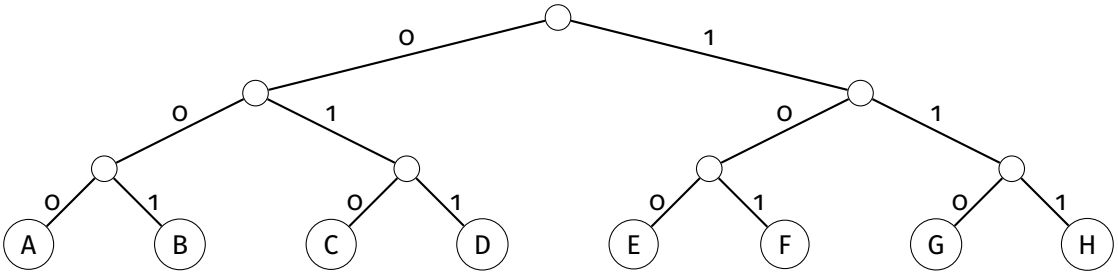


Abbildung 2: Codebaum für eine Kodierung fixer Länge.

Das Kodieren einer Nachricht mit Hilfe dieses Codebaums funktioniert analog zum Morsebaum, den wir bereits kennen. Hier verwenden wir statt Punkte und Striche halt einfach Nullen uns Einsen.

Uns fällt auf, dass die Code-Wörter in diesem Beispiel nur in den Blättern des Codebaumes vorkommen. Daraus lässt sich leicht ableiten, dass diese Kodierung eindeutig dekodierbar ist: Auf dem Weg von der Wurzel zu einem Code-Wort werden wir bei dieser Anordnung niemals auf ein anderes Code-Wort treffen, es gibt also keinerlei Verwechslungsgefahr!

Kodierungen, bei denen man auf dem Weg von der Wurzel zu einem Code-Wort niemals an einem anderen Code-Wort vorbei kommt, haben einen speziellen Namen: Sie sind **präfixfrei**.

Präfixfreie Kodierungen

Eine Kodierung heisst **präfixfrei**, wenn keines ihrer Code-Wörter ein Präfix eines anderen Code-Wortes ist. In anderen Worten: Kein Code-Wort bildet den Anfang eines anderen Code-Wortes.
Präfixfreie Kodierungen sind immer **eindeutig dekodierbar**.

? Präfixfreie Pfade

Siehst du den Zusammenhang zwischen eindeutigen Pfaden von der Wurzel zu einem Code-Wort im Codebaum und der Präfixfreiheit einer Kodierung?

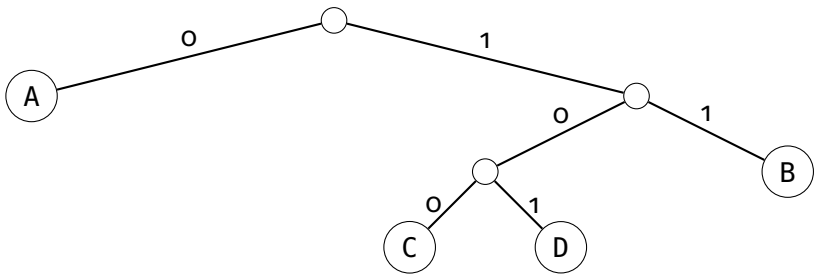
Beispiel – Präfixfreie Kodierungen

Kodierungen fixer Länge sind lediglich ein Beispiel von präfixfreien Kodierungen. Nicht alle präfixfreien Kodierungen haben eine fixe Code-Wortlänge!

Betrachte als Beispiel dafür eine Kodierung für das Alphabet `{A, B, C, D}` mit den vier Code-Wörtern {`0`, `11`, `100`, `101`}:

A	0
B	11
C	100
D	101

Diese Kodierung ist offensichtlich präfixfrei, da keines ihrer Code-Wörter ein Präfix eines anderen ist. Wenn wir den Codebaum dazu aufzeichnen, sehen wir auch, dass wir auf keinem Weg zu einem Code-Wort an einem anderen vorbeikommen:



Aufgabe 34 – Präfixfreie Kodierungen

Bestimme für die folgenden Kodierungen, ob sie präfixfrei sind.

(a) {0, 01, 11}

(b) {00, 01, 10, 11}

(c) {0, 100, 110, 1010, 1110}

(d) {10, 11, 000, 100, 1001}

Aufgabe 35 – Präfixfrei und selbstgemacht

Finde eine präfixfreie für ein Alphabet, welches aus den 6 Buchstaben {A, B, C, D, E, F} besteht. Gib dafür die entsprechenden Code-Wörter an und zeichne den Codebaum auf.

Aufgabe 36 (⋆) – Nicht präfixfrei und trotzdem eindeutig?

Wir haben bei der Definition von präfixfreien Kodierungen gesagt, dass alle präfixfreien Kodierungen eindeutig sind. Gilt die Umkehrung ebenfalls? Sind alle eindeutigen Kodierungen auch präfixfreie Kodierungen?

Tipp: Betrachte die letzte Kodierung in Aufgabe 34. Ist diese präfixfrei? Ist sie eindeutig? Findest du eine Beispielnachricht, welche nicht eindeutig dekodierbar ist?

Aufgabe 37 – UTF-8

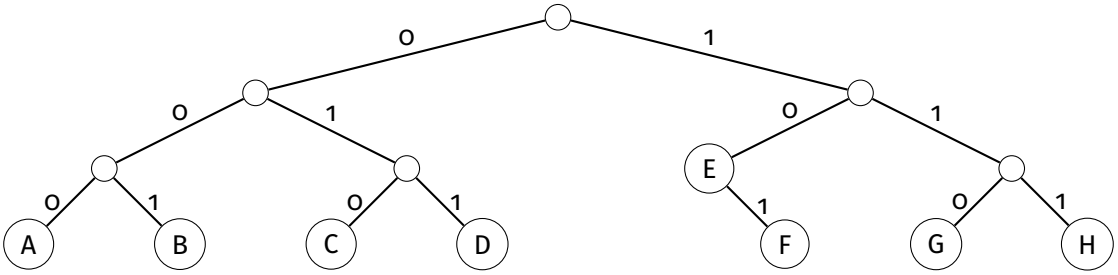
Erinnerst du dich an die UTF-8 Kodierung aus dem letzten Kapitel? Diese ist bekanntlich keine Kodierung fixer Länge, aber sie ist trotzdem eindeutig[7]! Kannst du nun begründen, warum dies so ist?

[7] Hoffentlich auch, sonst hätten wir ein riesiges Problem…

Die Huffman-Kodierung

Wir kennen nun eine Art von Kodierungen, die eindeutig dekodierbar sind, ohne dass dabei alle Code-Wörter gleich lang sein müssen. Wir wollen mit diesem Wissen unsere Kodierung aus Abbildung nun so umbauen, dass wir für viel verwendete Zeichen kürzere Code-Wörter erhalten, ohne aber dabei die eindeutige Dekodierbarkeit zu verlieren!

Nehmen wir an, der Buchstabe E kommt in zu kodierenden Texten viel häufiger vor als die restlichen Buchstaben. Ein kürzeres Code-Wort wäre hier also durchaus wünschenswert! Wir können dies erreichen, indem wir es einfach näher bei der Wurzel platzieren. Je näher sich ein Code-Wort an der Wurzel befindet, desto kürzer wird der Pfad dorthin, und desto kürzer auch das Code-Wort. Logisch, oder? Ein erster Versuch könnte also wie folgt aussehen: Wir schieben das Code-Wort für E eine Ebene nach oben und erhalten somit folgenden Codebaum:



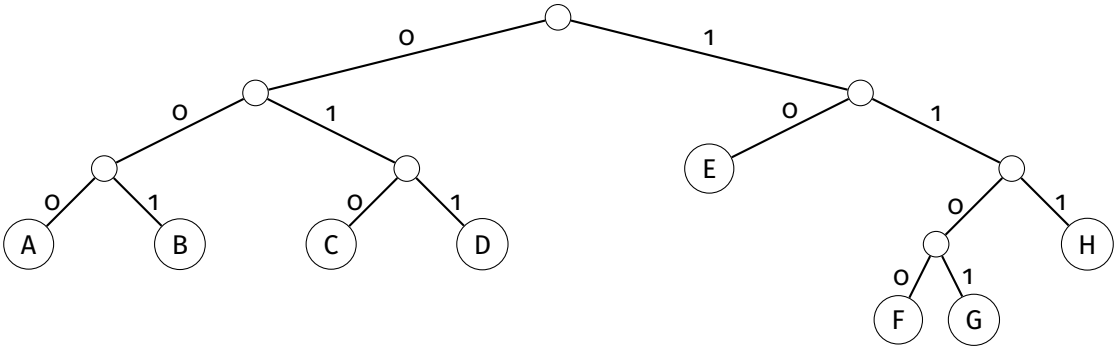
Jetzt haben wir aber wieder ein Dekodierungsproblem: Das Code-Wort für E (`10`) ist ein Präfix des Code-Wortes von F (`101`) - und somit ist unsere Kodierung nicht mehr präfixfrei[8]. Wenn wir unsere eben vorgenommene Optimierung beibehalten möchten, bleibt uns nichts anderes übrig, als das F an einem anderen Ort im Baum zu platzieren...

? Bäume verdrehen

> Wo würdest du das Code-Wort für F platzieren? Kannst du es einfach als Kind an ein bisheriges Blatt anhängen?
>
> Wenn du eine Lösung gefunden hast, zeichne den entsprechenden Codebaum auf!

[8] Das sagt zwar alleine noch nichts über die Eindeutigkeit aus, in diesem Falle resultiert es aber tatsächlich in einer Kodierung, die nicht eindeutig dekodierbar ist. Die Nachricht `1011010` können wir entweder als FEE oder als EGE auffassen.

Eine Möglichkeit, wie wir den Baum umgestalten können, zeigt folgende Abbildung:



Diese Kodierung ist wieder präfixfrei und somit eindeutig dekodierbar. Das Code-Wort für E ist nun kürzer als die restlichen, allerdings mussten wir zwei Code-Wörter (in diesem Beispiel F und G) dafür verlängern. Wenn wir aber davon ausgehen, dass der Buchstabe E öfter verwendet wird als die Buchstaben F und G zusammen, so sparen wir im Schnitt doch einige Bits beim Kodieren von Nachrichten.

Das gleiche Prinzip könnten wir jetzt natürlich für ein anderes Zeichen anwenden. Wenn wir annehmen, dass der Buchstabe A öfter verwendet wird als andere, so könnten wir auf der linken Seite des Baums eine ähnliche Umordnung vornehmen. Aber wie nahe sollen wir die häufig verwendeten Knoten (Code-Wörter) an die Wurzel schieben? Welche Zeichen opfern wir dafür, ordnen ihnen also längere Code-Wörter zu? Wo liegt das Optimum? Gibt es eine Art Rezept dafür? Fragen über Fragen… Bevor wir diese beantworten, lassen wir dich selbst probieren!

Aufgabe 38 – Codebäume selber bauen

In dieser Aufgabe wirst du eine Kodierung für die Zeichen □, ◯, △ und ★ erstellen. Verwende dabei das Computeralphabet {0,1} für deine Kodierung.

(a) Wie würdest du diese Zeichen mit einer Kodierung fixer Länge kodieren?

(b) Du weisst, dass □ häufiger verwendet wird als ◯, ◯ häufiger als △, und △ wiederum häufiger als ★. Wie würdest du diese vier Zeichen kodieren, wenn du die gegebenen Häufigkeiten ausnutzen möchtest? Achte dabei darauf, dass deine Kodierung eindeutig bleibt! Zeichne den entsprechenden Codebaum! Wie bist du dabei vorgegangen?

(c) Kodiere die Nachricht □◯△□◯□◯□□★□△□◯□◯□ zuerst mit deiner Lösung aus (a) und danach mit jener aus (b).

(d) Wie viele Bits sparst du mit deiner Kodierung aus (b) gegenüber derjenigen aus (a) ein, wenn du die obige Nachricht kodierst?

(e) Wie sieht es bei der Nachricht □□□□◯◯◯△△★ aus?

Du hast wahrscheinlich festgestellt, dass das Erstellen einer eindeutigen Kodierung nicht ganz so einfach ist, wie es auf den ersten Blick scheint. Erst recht, wenn man zudem noch die Häufigkeiten der zu kodierenden Zeichen berücksichtigen soll. Wie sieht eine optimale Kodierung aus? Wie spart man am meisten Bits, wenn man Zeichen um Zeichen kodiert?

Glücklicherweise hat jemand diese Frage schon beantwortet! David Huffman hat bereits 1952 folgendes Vorgehen vorgeschlagen:

Huffman-Kodierung

Gegeben sei eine Menge von zu kodierenden Zeichen sowie deren Häufigkeiten.

1. Notiere jedes Zeichen als einzelnen Knoten, inklusive dessen Häufigkeit, in einer Liste.
2. Sortiere diese Knotenliste anhand deren Häufigkeiten.
3. Entferne die beiden Knoten mit den kleinsten Häufigkeiten aus der Liste, und verbinde sie zu einem neuen Knoten, dessen Häufigkeit der Summe beider Häufigkeiten entspricht.
4. Füge diesen neu erhaltenen Knoten (mitsamt eventuellen Kindern) an der entsprechenden Stelle in der sortierten Liste ein.
5. Wiederhole diesen Vorgang von (3) aus, bis deine Liste nur noch aus einem einzelnen Knoten besteht.

Beschrifte nun die Kanten im eben kreierten Baum: Eine Abzweigung nach links entspricht einer `0`, eine Abzweigung nach rechts einer `1`. Die zu kodierenden Zeichen stehen nun in den Blättern des Baumes.
Den so erhaltenen Codebaum nennt man **Huffman-Baum**, die entsprechende Kodierung **Huffman-Kodierung**. Sie ist optimal[9] und präfixfrei, also eindeutig dekodierbar.

[9] Optimal, wenn wir Kodierungen betrachten, die Zeichen für Zeichen einzeln kodieren. Wenn wir auf die (sehr gute!) Idee kommen, zusammenhängende Textstellen als ganzes zu komprimieren, können wir eine viel bessere Kompression erreichen!

Beispiel – Huffman-Baum step by step

Das eben beschriebene Verfahren mag auf den ersten Blick abschreckend wirken. Du wirst aber bald merken, dass es gar nicht so kompliziert ist, wie es zunächst scheint! Wir wollen die Nachricht FALAFELAUFLAUF mit dem Huffman-Verfahren kodieren.

Als Erstes zählen wir also die Vorkommnisse der Buchstaben, um deren Häufigkeiten zu bestimmen: F kommt 4 Mal vor, A ebenfalls 4 Mal, L 3 Mal, E ein Mal und U zwei Mal.

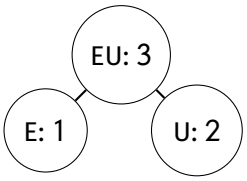
Nun sortieren wir diese Zeichen anhand der Häufigkeiten: A (4), F (4), L (3), U (2), E (1) und speichern uns diese Information als Knoten ab. Wir stellen die Liste unserer Knoten jeweils links dar, während wir den aktuellen Baum rechts zeichnen. Vor dem ersten Schritt haben wir lediglich eine Liste von Knoten, die wir nach Häufigkeit sortiert haben:

A: 4
F: 4
L: 3
U: 2
E: 1

Gemäss Schritt (3) der Huffman-Methode bestimmen wir die zwei am wenigsten verwendeten Knoten, in unserem Fall also E und U. Wir verbinden diese zu einem neuen Knoten, den wir hier EU nennen wollen. Die Häufigkeit dieses neuen Knotens ist die Summe der Häufigkeiten seiner Kinder, also $1 + 2 = 3$.

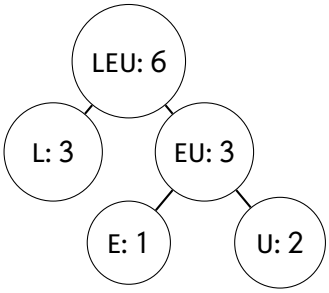
Wir streichen also die gerade verbundenen Knoten aus unserer Liste und fügen den neu erhaltenen Knoten ein:

A: 4
F: 4
L: 3
EU: 3
~~U: 2~~
~~E: 1~~



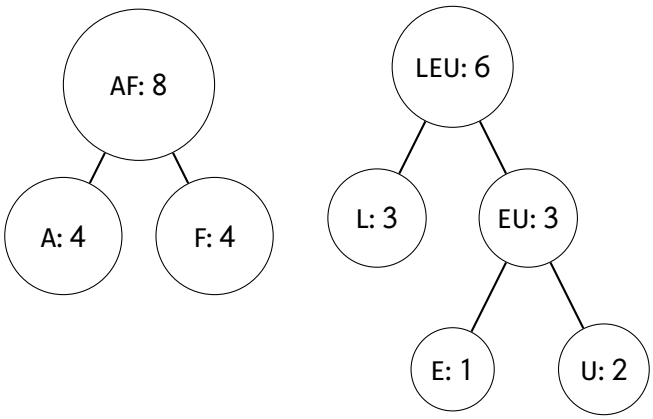
Wir wiederholen gemäss Rezept den gleichen Schritt, fügen also die kleinsten zwei Knoten in unserer Liste zu einem grösseren Knoten zusammen: L und EU wachsen zusammen zu einem Knoten, der nun die Häufigkeit 6 hat:

LEU: 6
A: 4
F: 4
~~L: 3~~
~~EU: 3~~
~~U: 2~~
~~E: 1~~



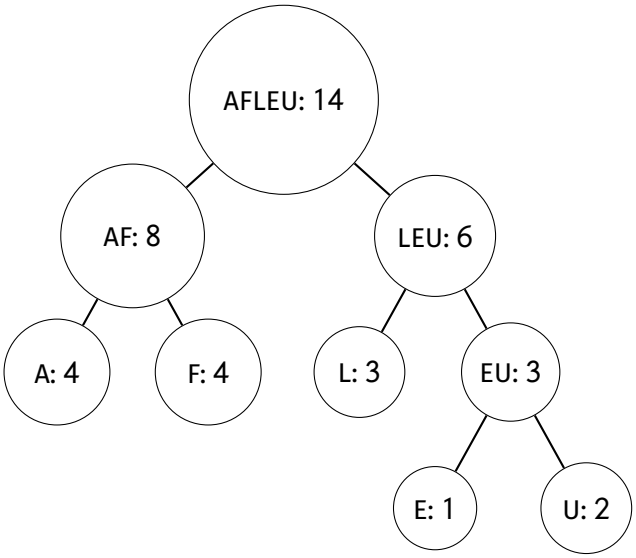
Als nächstes müssen wir also die Knoten A und F zusammenfassen, was in einem Knoten mit Häufigkeit 8 resultiert, welchen wir AF nennen:

AF: 8
LEU: 6
~~A: 4~~
~~F: 4~~
~~L: 3~~
~~EU: 3~~
~~U: 2~~
~~E: 1~~

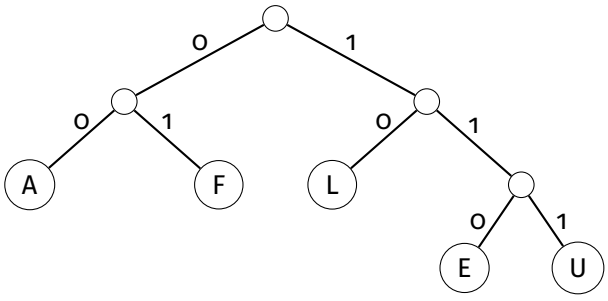


Zuletzt müssen wir die beiden übrig gebliebenen Knoten zusammenfassen, also verbinden wir AF und LEU:

AFLEU: 14
~~AF: 8~~
~~LEU: 6~~
~~A: 4~~
~~F: 4~~
~~L: 3~~
~~EU: 3~~
~~U: 2~~
~~E: 1~~

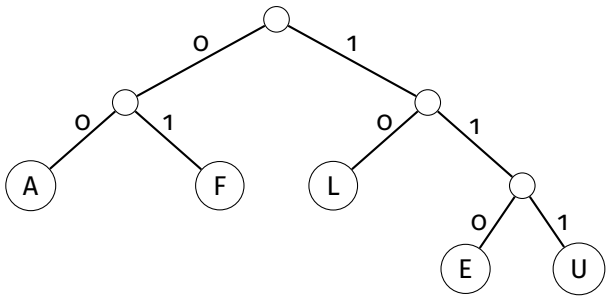


Nun sind wir gemäss Anleitung fertig, da wir nur noch einen Knoten in unserer Liste führen. Wir müssen lediglich noch die Kanten beschriften. Die Häufigkeiten und Beschriftungen der inneren Knoten brauchen wir zum Kodieren und Dekodieren nicht und lassen diese deshalb einfach weg. Schlussendlich resultiert folgender Codebaum:



Beispiel – Kodieren mit einem Huffman-Baum

Nun, da wir einen Codebaum für das Wort FALAFELAUFLAUF kreiert haben, wollen wir diese Nachricht natürlich auch noch Kodieren. Zur besseren Übersicht hier nochmals der resultierende Huffman-Baum aus dem obigen Beispiel:



Daraus können wir leicht die Code-Wörter für unsere fünf Zeichen ablesen: A wird als `00` kodiert, F als `01`, L wird zu `10`, E zu `110` und U schliesslich schreiben wir als `111`.

Wir bemerken, dass diese Kodierung tatsächlich präfixfrei ist, und die häufig verwendeten Buchstaben A und F kürzere Code-Wörter erhalten haben als die seltener verwendeten E und U.

Die kodierte Nachricht lautet `01 00 10 00 01 110 10 00 111 01 10 00 111 01`, das sind 31 Bits. Mit einer Kodierung fixer Länge hätten wir hier 3 Bits pro Zeichen benötigt, was zu total 42 Bits geführt hätte.

Aufgabe 39 – Huffman-Baum erstellen

Erstelle einen Huffman-Baum für das Wort ERDBEERWAEHE und kodiere es anschliessend mit deiner erhaltenen Kodierung.

Tipp: Wenn du mehrere Möglichkeiten hast um Knoten zusammen zu führen, darfst du frei entscheiden, welche zwei du auswählst!

(a) Wie lautet das kodierte Wort?

(b) Wie viele Bits benötigst du dafür?

(c) Wie viele würdest du bei einer Kodierung fixer Länge benötigen?

(d) Vergleiche deinen Huffman-Baum mit deinem Nachbarn oder deiner Nachbarin. Habt ihr den gleichen Baum erhalten?

(e) Erstelle Huffman-Bäume für die Wörter `KATAMARAN` und `DIALEKTFORSCHUNG`. Überlege dir zuvor jedoch, wie diese Bäume ungefähr aussehen werden!

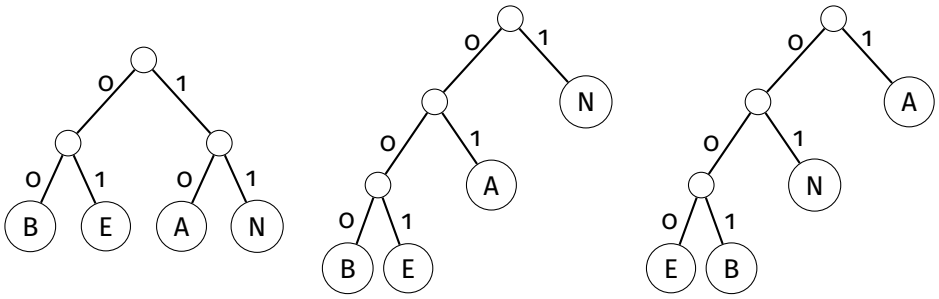
Huffman zum Anfassen

Unter diesem Link findest du ein kleines Tool, welches dir zu einem gegebenen Text die Häufigkeiten der einzelnen Zeichen ermittelt und anschliessend *einen* möglichen Huffman-Baum erstellt.

Ein Wort - viele mögliche Huffman-Bäume!

Ist die Huffman-Kodierung für ein bestimmtes Wort eindeutig? Aufgabe 39 hat dir hoffentlich gezeigt, dass dies nicht so ist. Erstens können wir die Kanten des Codebaums schlussendlich anders beschriften, z.B. eine 1 wählen für Abzweiger nach rechts und eine 0 für eine nach links.

Weiter gibt es auch Fälle, in denen wir eine Wahl haben, welche Knoten wir als nächstes zusammenführen möchten. Beim Kodieren des Wortes BANANE beispielsweise haben wir nach dem ersten Zusammenführen der Knoten B und E die Situation, dass wir drei Knoten in unserer Liste haben, die alle Häufigkeit 2 haben: BE, N und E. Welche Knoten wir als erstes auswählen, ist uns überlassen. Alle resultierenden Bäume sind optimal und kodieren das entsprechende Wort mit so wenig Bits wie möglich. Überzeuge dich selbst, indem du mit den folgenden drei möglichen Huffman-Bäumen das Wort BANANE kodierst:



Die kodierte Nachricht unterscheidet sich aber natürlich je nach Huffman-Baum. Deshalb ist es wichtig, dass wir beim Verschicken oder Abspeichern einer mit einem Huffman-Baum kodierten Nachricht immer den verwendeten Codebaum mitliefern - das Dekodieren der Nachricht wäre sonst nicht möglich! Bei sehr kleinen Nachrichten, wie wir sie in diesem Kapitel als Beispiele oder Aufgaben behandeln, ist dieser Overhead natürlich nicht praktikabel, sogar kontraproduktiv. Bei längeren Nachrichten allerdings fällt das Kodieren des Codebaums selbst nicht mehr so fest ins Gewicht und wir sparen alles in allem einiges an Speicherplatz oder Übertragungszeit!

Wieso die Huffman-Kodierung genau optimal ist, wollen wir an dieser Stelle nicht formal aufzeigen. Wir können aber sehr wohl unsere Intuition ein bisschen stärken, weshalb denn das Huffman-Verfahren eine gute und präfixfreie Kodierung liefert! Letzteres ist schnell geklärt: Alle zu kodierenden Zeichen finden sich in den Blättern des Huffman-Baums wieder. Die inneren Knoten sind nur Hilfskonstrukte, die wir während dem Aufbau benötigen. Auf dem Kodierweg von der Wurzel zu einem Blatt kommen wir in einem Baum sicherlich nicht an einem anderen Blatt vorbei, die Kodierung ist also sicherlich präfixfrei.

Weiter bemerken wir, dass Zeichen mit hoher Häufigkeit erst später während dem Verfahren in den finalen Codebaum eingepflegt werden. Und da wir den Baum von unten nach oben aufbauen bedeutet das ja, dass sie näher an der Wurzel liegen als solche Zeichen, die früher in den Baum einfliessen. Die Nähe zur Wurzel hat natürlich eine direkte Auswirkung auf die Länge des resultierenden Code-Wortes: Je weniger Zwischenstationen wir auf dem Weg durch den Baum zurücklegen müssen, desto kürzer ist das Code-Wort!

Aufgabe 40 – Gedanken zum Huffman-Baum

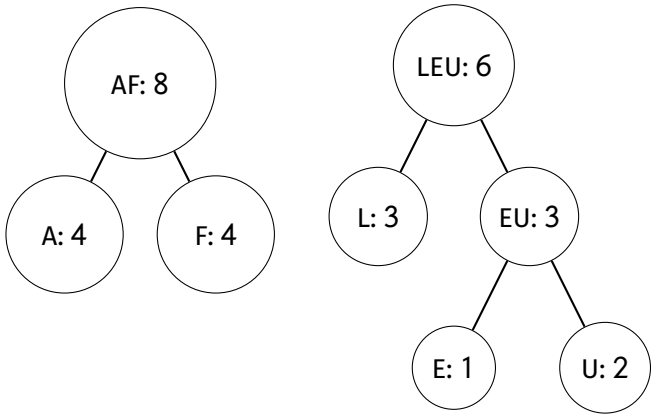
Zum Abschluss sollst du dir ein paar Gedanken zu Huffman-Bäumen machen:

(a) Kann ein Huffman-Baum in einer Kodierung fixer Länge resultieren? Wenn ja, in welchen Fällen?

(b) Ist es möglich, dass bei einer Huffman-Kodierung ein oder mehrere Code-Wörter länger sind, als dies bei einer entsprechenden Kodierung fixer Länge der Fall wäre? Wieso?

(c) Betrachte die inneren Knoten eines Huffman-Baums, der sich im Aufbau befindet:

AF: 8
LEU: 6
~~A: 4~~
~~F: 4~~
~~L: 3~~
~~EU: 3~~
~~U: 2~~
~~E: 1~~



Du siehst: so ein inneren Knoten ist immer grösser als seine einzelnen Kinder. Wieso ist das so? Was sagt die Häufigkeit eines inneren Knoten über die Häufigkeiten seiner Kinderknoten aus?

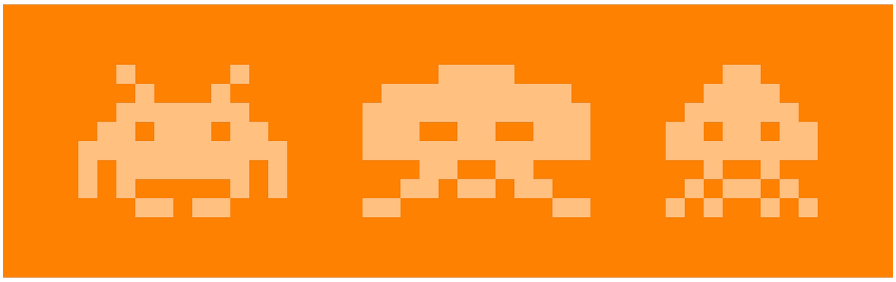
(d) Ist ein Huffman-Baum immer schön balanciert? Sprich: Ist die Summe der Häufigkeiten des Teilbaums auf der linken Seite der Wurzel immer gleich gross wie diejenige auf der rechten Seite der Wurzel? Findest du ein Beispiel, bei welchem das nicht der Fall ist?

Zusammenfassung

In diesem Kapitel haben wir gelernt, dass die Code-Wörter einer Kodierung nicht zwingend alle gleich lang sein müssen, um eine eindeutige Dekodierbarkeit zu gewährleisten. Ein Beispiel einer solchen Kodierung variabler Länge sind die **präfixfreien Kodierungen**, bei denen kein Code-Wort den Anfang eines anderen bildet. In der **Darstellung als Codebaum** erkennt man solche präfixfreien Kodierungen daran, dass ihre Zeichen nur in den Blättern des Baumes stehen, wodurch man beim Kodieren oder Dekodieren mit Hilfe des Codebaums von der Wurzel her nur an genau einem Zeichen vorbeikommt.

Um Speicherplatz oder Übertragungszeit zu sparen, können wir bei solchen Kodierungen **häufig verwendeten Zeichen kürzere Code-Wörter zuweisen**, während wir bei selten benötigten Zeichen etwas längere Code-Wörter in Kauf nehmen. Um eine solche Kodierung unter Berücksichtigung der Häufigkeiten der Zeichen zu erstellen, haben wir das Huffman-Verfahren kennen gelernt. Eine daraus resultierende **Huffman-Kodierung ist optimal** (sprich es existiert keine, die kürzere kodierten Nachrichten ergibt) und präfixfrei. Da der Huffman-Baum für ein gegebenes Wort nicht zwingend eindeutig ist, müssen wir aber den verwendeten Codebaum immer mit der kodierten Nachricht zusammen abspeichern oder verschicken.

Digitale Bilder

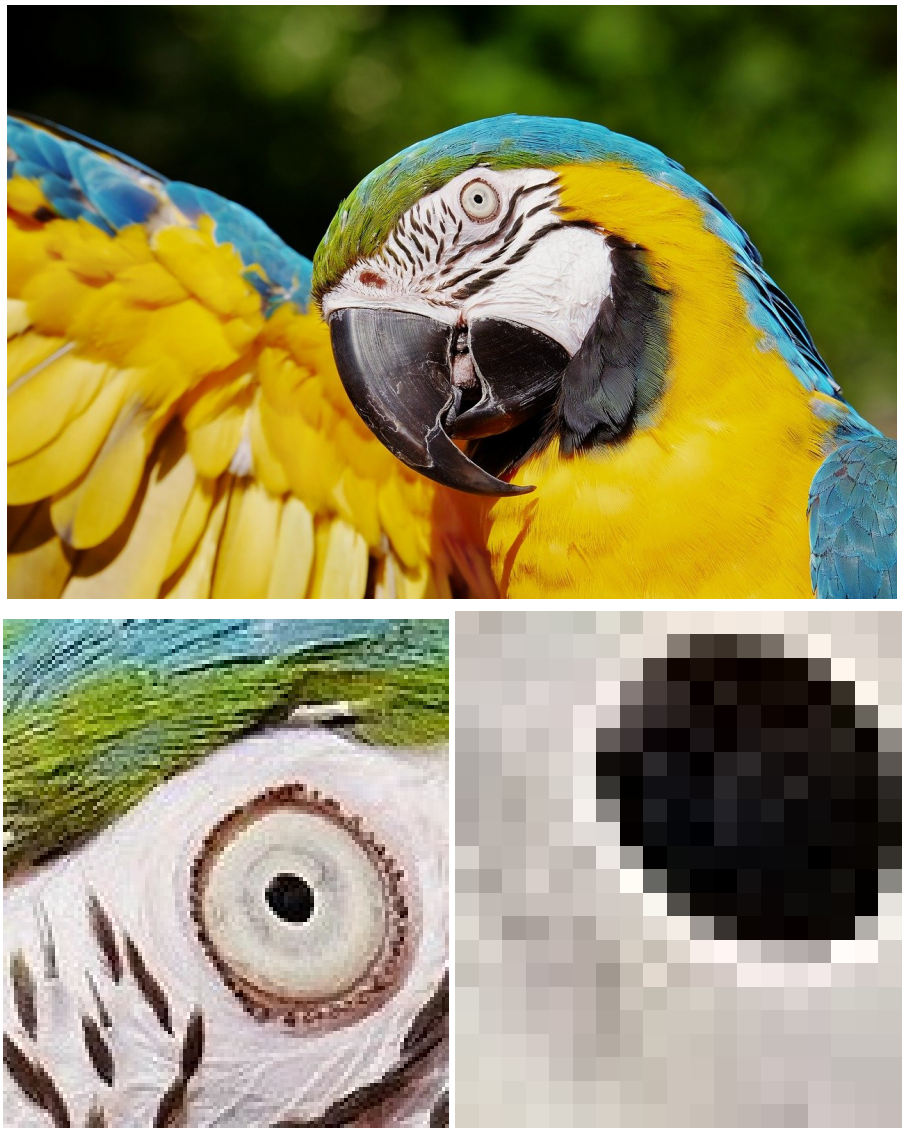


In diesem Kapitel befassen wir uns mit digitalen Bildern. Woraus bestehen diese, und wie werden diese auf einem Computer gespeichert? Und wie landen sie schliesslich farbig ausgedruckt auf einem Blatt Papier, oder digital auf deinem Monitor?

Pixel

Im Allgemeinen unterscheiden wir zwei Arten von digitalen Bildern: **Vektorgrafiken** und **Rasterbilder**. Bei ersteren wird der Inhalt einer Grafik durch geometrische Figuren wie Kreise, Polygone oder Pfade beschrieben und wird häufig für Logos oder Diagramme verwendet. Wir beschränken uns hier aber auf Rasterbilder.

Digitale Rasterbilder bestehen aus (sehr) vielen Bildpunkten, so genannten Pixeln, die rasterförmig angeordnet sind und denen eine bestimmte Farbe zugeordnet ist.



Ein Ara Papagei.
Oben: Gesamtansicht
Unten: vergrösserter bzw. stark vergrösserter Ausschnitt der Augenpartie

Pixel und Auflösung

Ein **Pixel** (von engl. picture element) gibt die Farbinformation eines Bildes an einer bestimmten Stelle wieder.
Die Anzahl Pixel in einem Bild nennt man **Auflösung** eines Bildes.

Die Auflösung geben wir oft auch als **Breite x Höhe** an. Je höher die Auflösung, desto schärfer erscheint das Bild, da mit mehr Pixel feinere Konturen und Details abgebildet werden können.

Beispiel – Auflösung

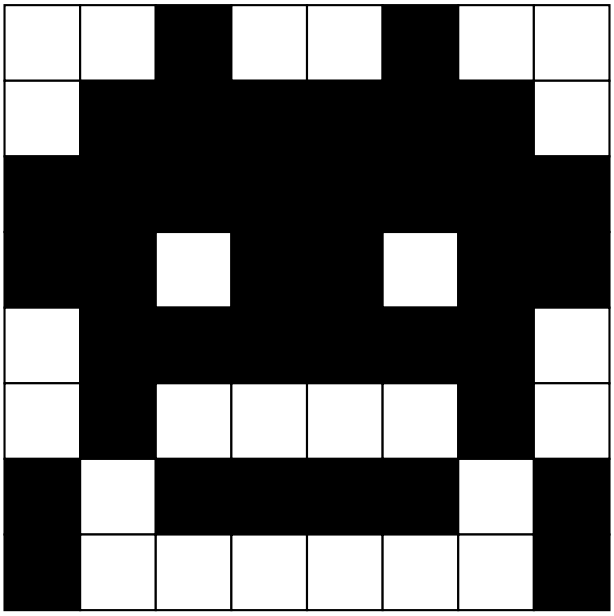
Ein Bild, welches 5000 Pixel breit und 2000 Pixel hoch ist, hat eine Auflösung von 5000 x 2000 Pixel, oder anders ausgedrückt 10 Megapixel (10 Millionen Pixel).

Aufgabe 41 – Auflösung

Ein Bild hat eine Auflösung von 800 Pixel. Du weisst, dass es 40 Pixel breit ist. Wie hoch ist es?

Monochrone Bitmaps

Die erste Art von Rasterbilder, die wir genauer anschauen wollen, sind **monochrome Bitmaps**. Der Begriff *monochrom* kommt aus dem Griechischen und bedeutet *einfarbig*. Wir verstehen darunter Rasterbilder, die nur aus Abstufungen einer Farbe bestehen: Schwarz und Weiss. Betrachte dazu das folgende Beispielbild eines kleinen Pixelmonsters:



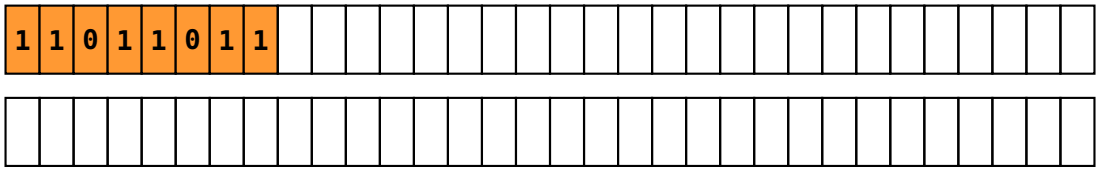
Das Bild hat eine Auflösung von 8 x 8 Pixel, und jedes Pixel ist entweder schwarz oder weiss. Wie können wir dieses Bild in binärer Form auf dem Computer darstellen?

Der erste Schritt, den wir dafür unternehmen, ist folgender: Wir repräsentieren jedes Pixel durch 1 Bit. Hat dieses den Wert 1, so ist das Pixel weiss (*Licht an*), bei einer 0 ist das Pixel schwarz (*Licht aus*):

1	1	0	1	1	0	1	1
1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0
1	0	0	0	0	0	0	1
1	0	1	1	1	1	0	1
0	1	0	0	0	0	1	0
0	1	1	1	1	1	1	0

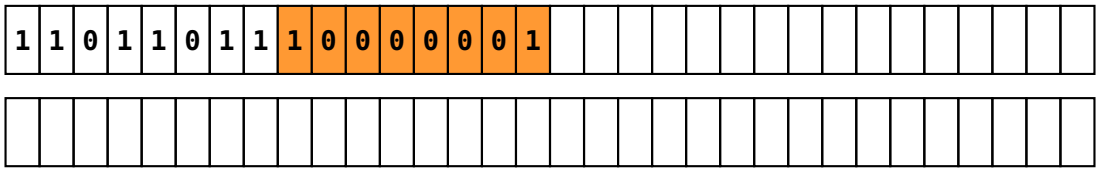
Leider ist der Arbeitsspeicher eines Computers nicht zweidimensional aufgebaut, wir müssen also diese Pixel als eine eindimensionale Folge von Bits darstellen können. Der naheliegendste Ansatz ist, diese einfach Reihe für Reihe aneinander zu knüpfen. Wir beginnen dafür mit der ersten Reihe:

1	1	0	1	1	0	1	1
1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0
1	0	0	0	0	0	0	1
1	0	1	1	1	1	0	1
0	1	0	0	0	0	1	0
0	1	1	1	1	1	1	0



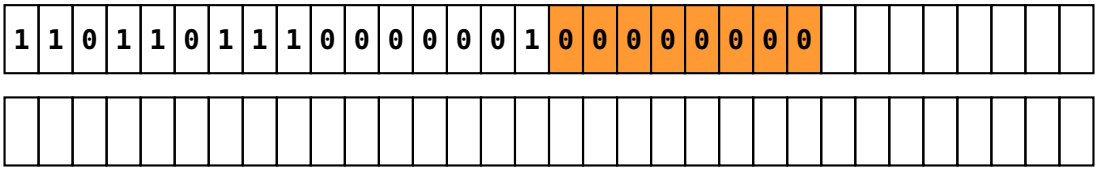
Dahinter fügen wir nun die zweite Reihe an:

1	1	0	1	1	0	1	1
1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0
1	0	0	0	0	0	0	1
1	0	1	1	1	1	0	1
0	1	0	0	0	0	1	0
0	1	1	1	1	1	1	0

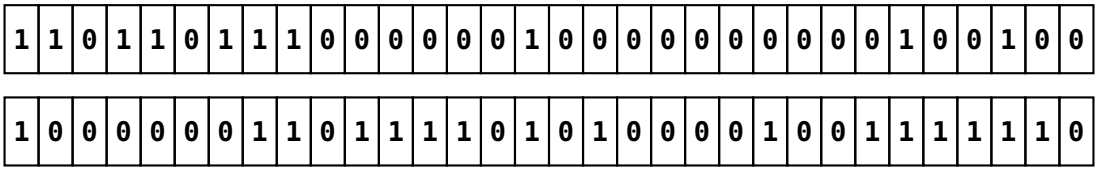


Und nun die dritte Zeile:

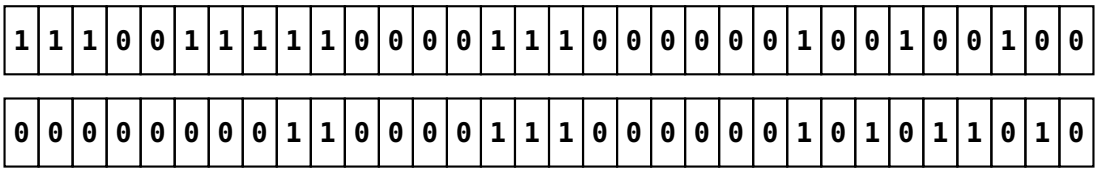
1	1	0	1	1	0	1	1
1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0
1	0	0	0	0	0	0	1
1	0	1	1	1	1	0	1
0	1	0	0	0	0	1	0
0	1	1	1	1	1	1	0



Du ahnst sicher, wie es weiter geht. Schlussendlich erhalten wir die folgende Darstellung:

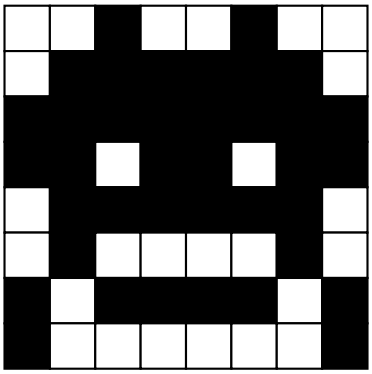


Sind wir nun fertig? Leider noch nicht ganz…Um dies zu verdeutlichen, versuchen wir, eine andere solche Bitfolge rückwärts, also von der eindimensionalen in die zweidimensionale Darstellung zu übersetzen:

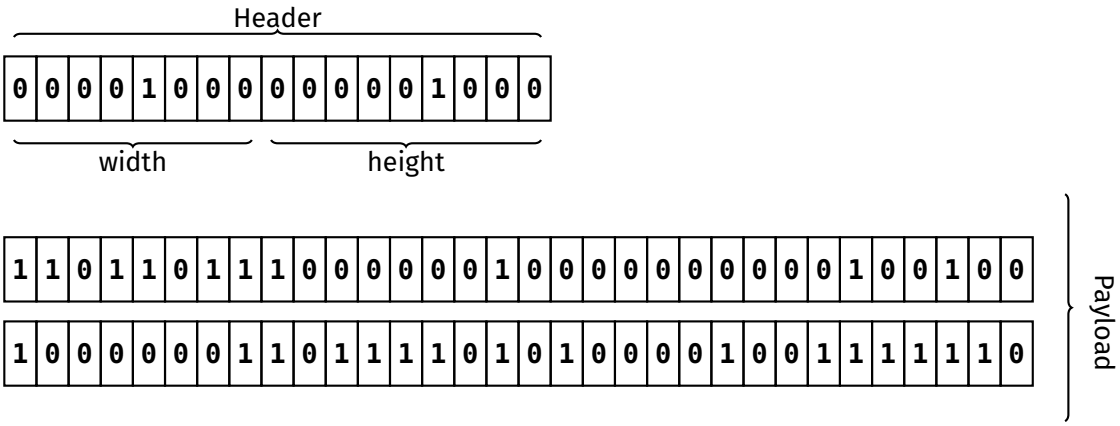


Was fehlt uns? Genau, wir wissen gar nicht, welche Dimensionen das gewünschte Bild hat. Alles, was wir wissen, ist, dass das Bild aus 64 Pixel besteht. Sind es 8x8 Pixel? 4x16 Pixel? 2x32 Pixel? 1x64 Pixel? Oder umgekehrt? Wir wissen es nicht. Diese Information müssen wir also ebenfalls mitkodieren! Wir tun dies in einem so genannten **Header**. Dieser beinhaltet Metadaten des Bildes, wie beispielsweise eben Breite und Höhe. In der Praxis werden noch viele weitere Eigenschaften angegeben, wir beschränken uns hier der Einfachheit halber auf diese zwei. Wir definieren unseren Header also als Bitfolge der Länge 16, wobei die ersten 8 Bits die Breite und die zweiten 8 Bits die Länge repräsentieren.

In unserem Beispiel ist sowohl die Höhe als auch die Breite 8 Pixel, die Binärdarstellung dafür lautet `00001000`. Als Header schreiben wir deshalb `00001000 00001000`. Nach dem Header folgt der **Payload**, also die Folge von Pixeldaten. Unser Pixelmonster



wird also wie folgt repräsentiert:



Mit dieser Information können wir also folgende Bitfolge als Bild interpretieren:

0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0

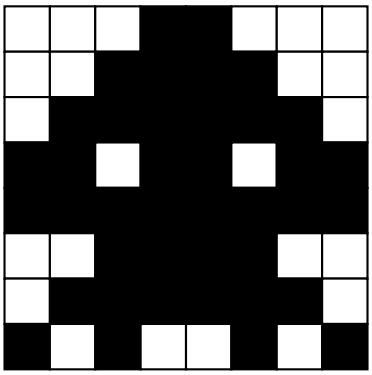
1	1	1	0	0	1	1	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0

0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	1	0	1	1	0	1	0

Der Header verrät uns, dass sowohl die Höhe als auch die Breite 8 Pixel beträgt. Alles, was wir nun machen müssen, ist, den Payload in das entsprechende 8x8-Raster einzutragen:

1	1	1	0	0	1	1	1
1	1	0	0	0	0	1	1
1	0	0	0	0	0	0	1
0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	1
1	0	0	0	0	0	0	1
0	1	0	1	1	0	1	0

Damit erhalten wir folgendes Bild:

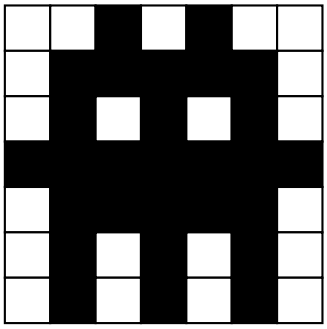


Aufgabe 42 – Header

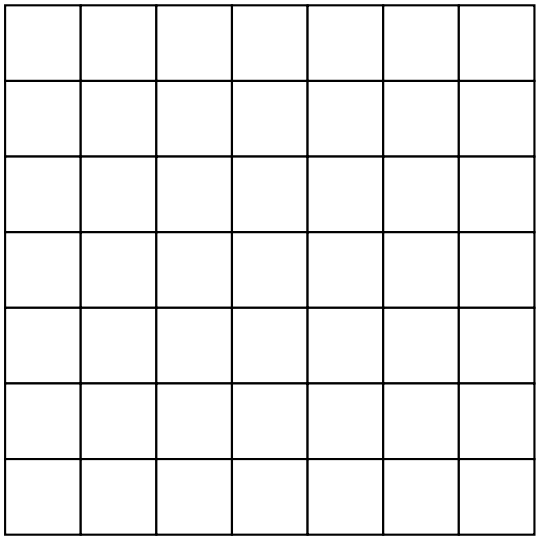
Was ist die maximale Grösse eines Bildes, das mit unserer Headerdefinition (8 Bits für die Breite, 8 Bits für die Höhe) repräsentiert werden kann?

Aufgabe 43 – More monsters

Betrachte die folgende Bitmap:



(a) Stelle diese zuerst in einem binären Grid dar, indem du die entsprechenden Bits in untenstehendes Raster einträgst:



(b) Wie wird diese repräsentierst du diese Bitmap als Bitfolge? Markiere in deiner Lösung, welche Bits zum Header gehören, und wo der Payload beginnt.

Aufgabe 44 – Zahl als Bild

Welche Zahl verbirgt sich hinter folgender monochromen Bitmap? Der Header ist hier wiederum 16 Bits lang und definiert die Breite und Höhe des Bildes.

0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	1

0	1	0	0	1	0	0	0	0	1	1	0	1	1	0

Aufgabe 45 – Custom Monster

Zeichne eine monochrome Bitmap, die maximal 10x10 Pixel gross ist. Kodiere sie danach als Bitfolge (inkl. Header!) und übergib das Resultat deiner Banknachbarin oder deinem Banknachbarn zum Dekodieren. Schafft ihr es, eure eigenen Bilder korrekt zu kodieren und die vom Partner erhaltene Bitfolge korrekt zu dekodieren?

Graustufen

Bei monochronen Bitmaps haben wir jedes Pixel durch ein einziges Bit repräsentiert und somit nur zwei Farben bzw. Zustände unterscheiden können: weiss oder schwarz, *Licht an* oder *Licht aus*. Was müssen wir nun berücksichtigen, wenn wir pro Pixel mehr als nur zwei Farben zur Auswahl haben wollen? Richtig, wir brauchen pro Pixel mehr als nur ein Bit, um diese Abstufungen zu unterscheiden!

Mit 2 Bits pro Pixel können wir 4 verschiedene Kombinationen unterscheiden: `00`, `01`, `10` und `11`. Ein Monitor könnte diese Pixelwerte beispielsweise als *schwarz*, *dunkelgrau*, *hellgrau* beziehungsweise *weiss* auffassen. Oder anders ausgedrückt: verschiedene Graustufen. Je tiefer die Zahl, desto dunkler die Farbe.

Wenn wir jeden Pixelwert mit 3 Bits kodieren, können wir bereits 8 verschiedene Abstufungen unterscheiden. Und mit jedem zusätzlichen Bit verdoppelt sich die Anzahl darstellbarer Graustufen! Diese Anzahl Bits, die wir pro Pixel zur Verfügung haben, nennt man auch **Farbtiefe**.

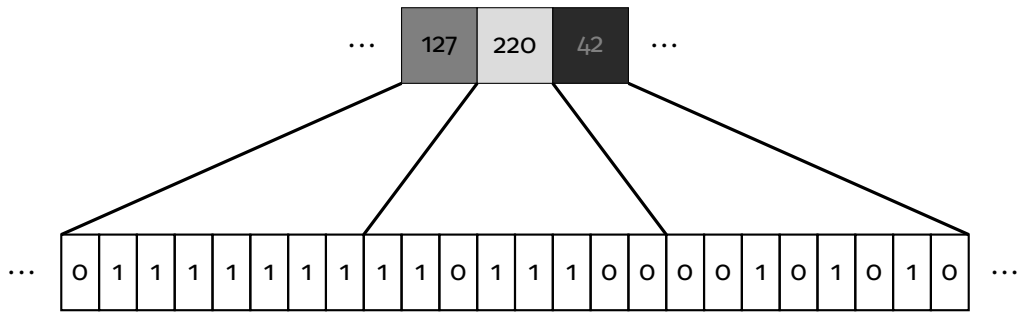
Farbtiefe

Die **Farbtiefe** eines Bildes bestimmt, wie viele verschiedene Farben ein Pixel annehmen kann. Je grösser die Farbtiefe, desto mehr Farben können dargestellt werden.
Die Farbtiefe wird meist in Bits gemessen.

Die meisten Graustufenbilder, die man in der Praxis antrifft, haben eine Farbtiefe von 8 Bit. Damit lassen sich also 256 verschiedene Grautöne beschreiben: von 0 (schwarz) bis 255 (weiss):

0	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	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Wenn wir ein solches Graustufenbild (mit einer Farbtiefe von 8 Bit) also als Bitfolge repräsentieren wollen, gehören immer 8 aufeinanderfolgende Bits zum gleichen Pixel und bestimmen dessen Grauton. Das folgende Schemabild verbildlicht dies:



Das erste Pixel aus dem Ausschnitt hat den Grauwert 127, was binär geschrieben `01111111` ist. Die nächsten 8 Bits definieren die Farbe des nächsten Pixels: `11011100` ergibt 220. Analog für das nächste Pixel: der Grauton 42 wird als `00101010` kodiert.

Du siehst: Je grösser die Farbtiefe, desto mehr Farben können wir darstellen. Allerdings wächst damit natürlich auch der Speicherplatzbedarf des Bildes!

Aufgabe 46 – Graustufen

(a) Wie viele Graustufen können wir mit 4 Bit darstellen?

(b) Was entspricht dabei der Farbe schwarz, was weiss?

(c) Wie sieht die Situation bei einer Farbtiefe von 5 Bit aus?

(d) Wie gross muss die Farbtiefe sein, wenn wir 128 verschiedene Graustufen darstellen wollen?

Beispiel – 4-Bit Graustufen

Wir betrachten ein Graustufenbild, welches eine Farbtiefe von 4 Bit hat. Das bedeutet, dass darin $2^4 = 16$ verschiedene Grautöne abgebildet werden können und jedes Pixel mit 4 Bits repräsentiert wird.

Als Bitfolge sieht das Bild wie folgt aus:

```
00000100000010000000000010010001101000101011001111
10001001101010111100110111101111
```

Als erstes teilen wir unser Bild in unser bekanntes Schema mit Header und Payload auf:

0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1

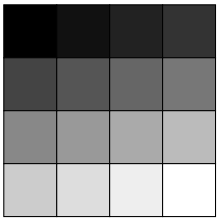
1	0	0	0	1	0	0	1	1	0	1	0	1	0	1	1	1	1	0	0	1	1	0	1	1	1	1	0	1	1	1	1

Der Header sagt uns, dass das Bild 4 Pixel breit und 4 Pixel hoch ist.

Danach gehen wir den Payload, also alles ausser dem Header, Pixel für Pixel durch. Jedes Pixel wird durch 4 Bit repräsentiert, weshalb wir jeweils 4 Bit zu einer Zahl zusammenfassen. In diesem Fall erhalten wir also

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Wir wissen, dass wir mit 4 Bits 16 verschiedene Grautöne darstellen können, mit den Werten von 0 (schwarz) bis 15 (weiss). Wenn wir dies in Graustufen ausdrücken, sieht das folglich wie folgt aus:



Aufgabe 47 – Farbtiefe im Header

Dir ist vielleicht aufgefallen, dass wir in unserem Header nirgens definieren, wie gross die Farbtiefe ist. Natürlich könnte man diese Information der Einfachheit halber auch noch hinzufügen. Die Frage ist aber: ist das in unserem Modell wirklich notwendig? Wie können wir die Farbtiefe auch ohne explizite Angabe ermitteln?

Aufgabe 48 – Graustufen dekodieren

Wir betrachten in dieser Aufgabe Graustufengrafiken mit einer Farbtiefe von 2 Bit, wir können damit also 4 Graustufen darstellen: schwarz (0), dunkelgrau (1), hellgrau (2) und weiss (3).

(a) Betrachte folgende Beispielgrafik, bei welcher die entsprechenden Pixelwerte (im Dezimalsystem) gegeben sind:

3	0	1
1	2	2
3	0	1

Skizziere, wie diese Bitmap etwa aussehen würde. Verwende dafür folgendes Raster:

(b) Wie sehen die einzelnen Pixelwerte in binärer Form aus? Fülle diese ins Raster ein:

(c) Wie sieht diese Bitmap als Bitfolge aus? Vergiss dabei den Header nicht!

Farbe ins Spiel bringen

Graustufen sind ja schön und recht, wir wollen nun aber etwas Farbe ins Spiel bringen. Dafür müssen wir aber zuerst zwei wichtige Farbmodelle unterscheiden: Das **CMYK**-Modell und das **RGB**-Modell

Das erste Modell, welches wir uns anschauen, heisst **CMYK**. Es wird unter anderem von Druckern verwendet. Bei CMYK handelt es sich um ein **subtraktives Farbmodell**. Das bedeutet, dass ein Mischen von zwei Farben in einer dunkleren Farbe resultiert. Das kannst du dir beispielsweise anhand von Wasserfarben verdeutlichen: Je mehr unterschiedliche Farben wir mischen, desto dunkler wird die resultierende Farbe, weil weniger Licht reflektiert wird. Wir wollen uns hier aber nicht mit den physikalischen Details beschäftigen.

Die Grundfarben des CMYK-Modells sind Cyan, Magenta, Gelb (Yellow), ergänzt durch eine Schlüsselfarbe (Key), meist schwarz. Das folgende Diagramm zeigt diese Grundfarben sowie deren Mischfarben rot, grün und blau. Die Mischung aller drei Grundfarben ergibt schwarz. Oft haben Drucker aber eine separate Tintenpatrone für diese Farbe. Dies einerseits, weil eine Mischung in der Praxis nicht das gewünscht saubere Resultat ergibt, andererseits um wertvolle Farbe zu sparen.

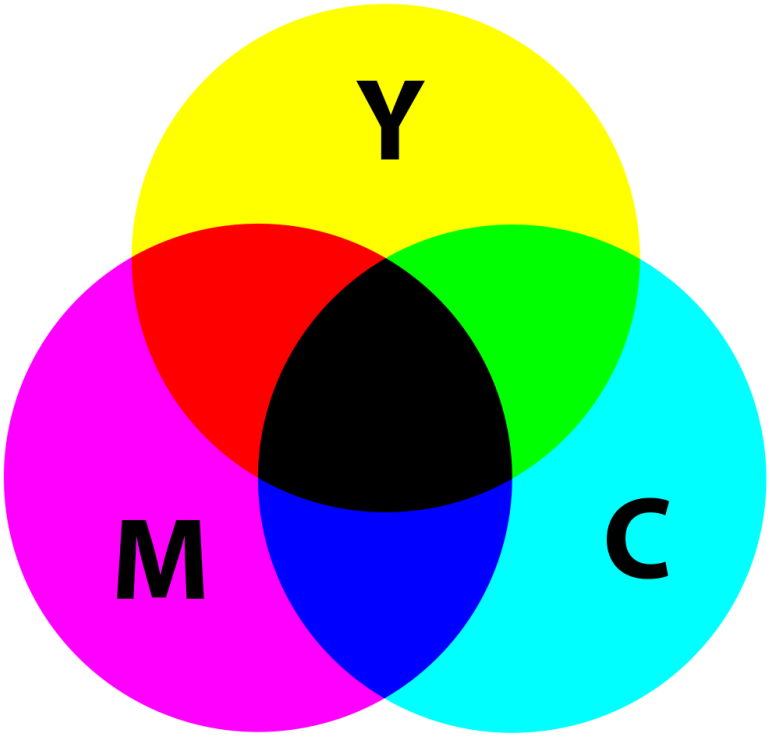


Abbildung 3: Die Drei Grundfarben des CMYK-Farbmodells und ihre primären Mischfarben.

Das zweite Modell, das wir näher betrachten wollen, ist das **RGB**-Modell. RGB steht dabei für rot (red), grün (green) und blau (blue) – die Grundfarben dieses Modells. Im Gegensatz zum CMYK-Modell ist das RGB-Modell aber kein subtraktives, sondern ein additives Farbmodell. Stell dir dazu drei Scheinwerfer vor, welche jeweils ihr Licht auf eine gemeinsame Fläche strahlen. Je mehr diese Scheinwerfer strahlen, desto mehr Licht gelangt in unser Auge, und desto heller nehmen wir die Farbe wahr.

Das folgende Diagramm zeigt dir die Grundfarben sowie deren primäre Mischfarben (magenta, gelb und cyan).

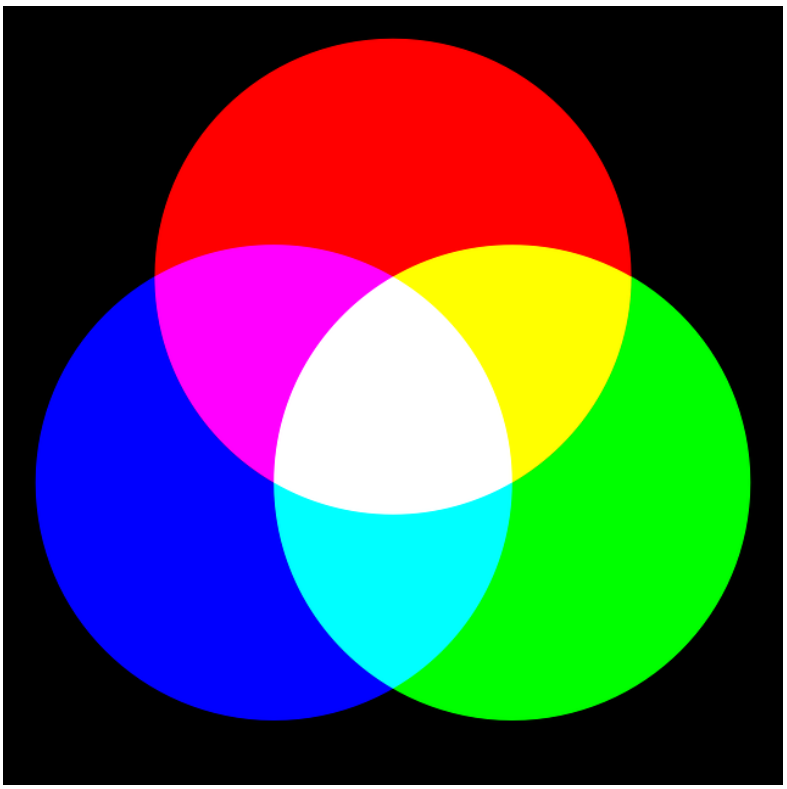


Abbildung 4: Die Drei Grundfarben des RGB-Farbmodells und ihre primären Mischfarben.

Das RGB-Modell ist jenes, welches von den allermeisten Bildschirmen verwendet wird. Die Oberfläche des Monitors besteht aus ganz vielen Punkten, wobei jeder dieser Punkte 3 Leuchten (meist LEDs) besitzt, die gemischt die gewünschte Farbe erzeugen. Beeindruckend, nicht?

Wie repräsentieren wir Farbbilder nun aber auf einem Computer? Zuerst müssen wir uns für ein Farbmodell entscheiden. Da sich das RGB-Modell für Monitore bestens eignet, ist dies auch unsere präferierte Lösung. Meist werden Farben für digitale Medien in diesem Format angegeben. Nun müssen wir noch eine Lösung finden, um die drei Farbkanäle (rot, grün, blau) pro Pixel zu definieren.

Die naheliegendste Lösung ist gleichzeitig auch die, welche in der Praxis verwendet wird: Wir fügen einfach die (binären) Werte der einzelnen Kanäle aneinander. Wie viele Bits wir dabei pro Farbkanal zur Verfügung haben, ist wiederum durch die Farbtiefe bestimmt: Eine Farbtiefe von 3 Bit beispielsweise heisst, dass wir für jeden Farbkanal 1 Bit zur Verfügung haben, also nur eine einzige Abstufung der entsprechenden Grundfarbe abbilden können. Damit lassen sich lediglich $2^3 = 8$ verschiedene Farben darstellen, nämlich die drei Grundfarben rot, grün und blau sowie deren Mischfarben.

Aufgabe 49 – 3-Bit RGB

Fülle die fehlenden Einträge mit Hilfe von Abbildung 4 in untenstehender Tabelle aus.

rot	grün	blau	resultierende Farbe
0	0	0	schwarz
0	0	1	blau
0	1	0	
0	1	1	
			rot
1	0	1	
			gelb
			weiss

Auch bei RGB-Bildern gilt: Je höher die Farbtiefe, desto mehr Farben können dargestellt werden. Wenn wir pro Farbkanal 2 Bits verwenden, können wir jeweils 4 Abstufungungen der Grundfarben miteinander mischen. Dies entspricht dann einer Farbtiefe von 6 Bit und erlaubt es uns, $2^6 = 64$ verschiedene Farben darzustellen.

Für RGB-Bilder wird heutzutage meist eine Farbtiefe von 24 Bit verwendet – wir haben also pro Farbkanal 8 Bits zur Verfügung, sprich 256 Abstufungen: 256 verschiedene Rottöne, 256 verschiedene Grüntöne und 256 verschiedene Blautöne. Gesamthaft ergibt dies $2^{24} = 16777216$ verschiedene Farben. Diese Konstellation nennt man auch **true colour** format.

Wenn wir diese drei Komponenten als Binärzahlen ausdrücken und aneinander reihen, erhalten wir pro Pixel eine Länge von 24 Bits, eben dem Wert der Farbtiefe. Ein Pixel wird also mit 24 Bits repräsentiert, wobei die ersten 8 den Rotanteil, die mittleren 8 den Grünanteil und die letzten 8 den Blauanteil bezeichnen.

Die folgende Grafik zeigt einen solchen Pixel mit den RGB-Werten 212, 142 und 51. Diese Mischung ergibt ein ockerfarbiges Orange und wird binär wie folgt repräsentiert:

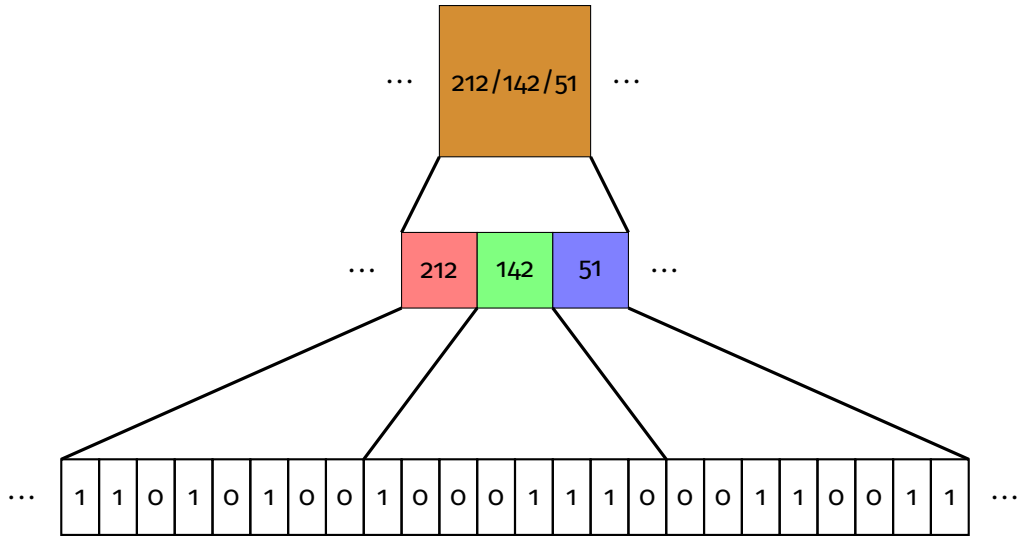


Abbildung 5: Schema zur Darstellung eines Pixels in einer RGB-Bitmap mit Farbtiefe einer Farbtiefe von 24 Bit.

Aufgabe 50 – RGB und Farbtiefe

Alice möchte für ein RGB-Bild pro Farbkanal gerne 4 verschiedene Farben darstellen können. Wie viele Bits benötigst du pro Farbkanal? Wie gross ist resultierende Farbtiefe des Bildes?
Wie sieht es aus, wenn Alice 32 Farben pro Kanal darstellen möchte?

Aufgabe 51 – Speicherplatz

Ein Bild hat die Auflösung 800 x 600 Pixel. Wenn du den Header für einmal ignorierst, vieviel Speicherplatz benötigt das Bild, wenn es sich dabei um

(a) eine monochrome Bitmap handelt?

(b) ein Graustufenbild mit Farbtiefe 4 Bit handelt?

(c) ein RGB-Bild mit Farbtiefe 12 Bit handelt?

(d) ein RGB-Bild mit 8 Bit pro Farbkanal handelt?

Abbildung 6 zeigt eindrücklich auf, welchen Einfluss die Farbtiefe auf die Darstellung eines Bildes hat. Bei einem Bit weniger pro Farbkanal können wir von blossem Auge kaum einen Unterschied ausmachen. Wenn wir die Farbtiefe halbieren, sehen wir bereits einige kleinere Unschärfen, vor allem in Bereichen mit vielen ähnlichen Farben.

Wenn wir schliesslich pro Farbkanal nur 1 Bit zur Verfügung haben, besteht das Bild nur aus den drei Grundfarben sowie deren Mischfarben.

24 Bit Farbtiefe
(8 Bits pro Kanal)

21 Bit Farbtiefe
(7 Bits pro Kanal)

12 Bit Farbtiefe
(4 Bits pro Kanal)

9 Bit Farbtiefe
(3 Bits pro Kanal)

6 Bit Farbtiefe
(2 Bits pro Kanal)

3 Bit Farbtiefe
(1 Bit pro Kanal)

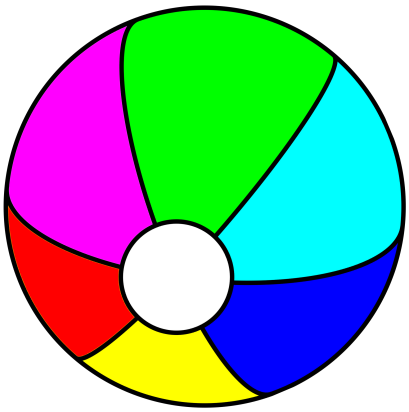
Abbildung 6: Ein RGB-Bild mit unterschiedlicher Farbtiefe

Aufgabe 52 – Missing channels

Betrachte die folgenden zwei Bilder, bei denen jeweils ein Farbkanal fehlt (sprich: ein Kanal hat jeweils immer den Wert 0). Findest du heraus, welcher Kanal das in welchem Bild ist? An was erkennst du das?

Aufgabe 53 – Defekter RGB-Monitor

Um besser für den Fernunterricht gerüstet zu sein, hast du dir einen neuen (RGB-) Monitor besorgt. Leider stellst du fest, dass bei diesem Monitor der **rote Farbkanal defekt** ist, die entsprechenden LEDs also kein Licht ausstrahlen.
Betrachte das Bild eines farbigen Strandballs:



Die Farben der Flächen im Uhrzeigersinn sind grün, cyan, blau, gelb, rot und magenta. Der weisse Kreis in der Mitte ist weiss, die Kontoren sind schwarz.
Wie würde dieses Bild auf deinem defekten Monitor dargestellt? Gib für jede Fläche und die Konturen an, ob und wie sie sich im Vergleich zum Originalbild verändert und wie sie auf deinem defekten Monitor aussieht!

Zusammenfassung

Digitale Rasterbilder besteht aus vielen **Pixeln**, die jeweils die Farbinformation an einem gewissen Punkt im Bild wiedergeben. Je mehr Pixel eine solche Rastergrafik enthält, desto grösser ist die **Auflösung** des Bildes.

In der digitalen Welt werden solche **Bitmaps** einfach durch eine Folge von Bits angegeben. Die ersten paar Bits nennt man **Header**, darin finden sich Meta-Informationen wie zum Beispiel die Dimensionen eines Bildes. Danach folgen die einzelnen Pixelwerte. Diesen Bereich nennt man **Payload**.

Das einfachste Beispiel von Bitmaps sind die **monochromen Bitmaps**, bei welchen jedes Pixel entweder schwarz oder weiss ist. Diese Zustände werden durch ein einziges Bit pro Pixel repräsentiert: 0 für schwarz und 1 für weiss.

Bei **Graustufenbildern** wird ein Pixel wiederum durch eine (binäre) Zahl dargestellt. Je grösser diese Zahl, desto heller ist die (Grau)Farbe, welche durch das Pixel abgebildet wird. Wir benötigen hier also im Allgemeinen mehr als 1 Bit pro Pixel um verschiedene Graustufen repräsentieren zu können.

Die Anzahl darstellbarer Farben oder Graustufen in einem Bild wird durch die **Farbtiefe** bestimmt. Dieses Mass wird oft in Bit angegeben: der Anzahl Bits, die es benötigt, um die gewünschte Anzahl Farben binär zu repräsentieren.

Das **RGB-Modell** ist das wohl bekannteste Farbmodell für digitale Farbbilder. Eine Farbe wird darin mit 3 Werten angegeben: Dem Rotwert, dem Grünwert und dem Blauwert. Im weit verbreiteten true-colour Format verwendet man pro Grundfarbe 8 Bits, die Farbtiefe ist also 24 Bit – und genau so viele Bits braucht es, um ein einzelnes Pixel binär zu repräsentieren. 8 Bits für den Rotwert, 8 für den Grünwert und 8 für den Blauwert.

Lösungen zu ausgewählten Aufgaben

Lösung zu Aufgabe 2

Die vier möglichen Bitfolgen der Länge 2 sind `00`, `01`, `10` und `11`.

Für eine Länge von 3 können wir an jede dieser Folge jeweils entweder eine `0` oder eine `1` anhängen, total ergibt das dann 8 mögliche Folgen. Analog erhalten wir aus diesen 8 Möglichkeiten deren 16 für die Länge 4.

Für n-Bit lange Folgen gibt es im Allgemeinen 2^n Möglichkeiten, da wir für jede Position aus 2 möglichen Zeichen wählen können.

Lösung zu Aufgabe 5

(a) Wir können Stäbchen der Längen 1, 5, 25, 125 etc. verwenden, also Stäbchen, deren Länge eine Potenz von 5 ist. Von jeder Sorte dürfen wir zwischen 0 und 4 Stück verwenden.

(b) Die Zahl 28 würden wir also wie folgt legen:

(c) Die Fünferzahl 123 besteht aus 3 Einern, 2 Fünfern und 1 Fünfundzwanziger. $1 \cdot 25 + 2 \cdot 5 + 3 \cdot 1 = 38$.

Lösung zu Aufgabe 7

(a) 2 (1 Zweier und 0 Einer)

(b) 6 (1 Vierer, 1 Zweier und 0 Einer)

(c) 7

(d) 8

(e) 10

(f) 17

(g) 136

Lösung zu Aufgabe 8

(a) $111_2 = 7$

(b) $1111_2 = 15$

(c) $11111_2 = 31$

(d) Die grösste Binärzahl der Länge n sind einfach n Einsen. Ihr Wert beträgt $2^n - 1$

Lösung zu Aufgabe 10

(a) Die grösste Zweierpotenz, die in 3 reinpasst, ist die 2. Danach bleibt noch ein Einer übrig, wir erhalten also 11_2.

(b) 110_2

(c) 1010_2

(d) In die 11 passt als grösste Zweierpotenz $2^3 = 8$ rein. 4er haben danach keine mehr Platz, aber ein 2er und ein 1er: 1011_2

(e) 11011_2

(f) 101000_2

(g) $128 = 2^7$ ist gerade eine Zweierpotenz. Somit brauchen wir einen 128er, und den Rest füllen wir mit Nullen auf: 10000000_2

Lösung zu Aufgabe 11

(a) 11_2

(b)

6	:	2	=	3	Rest 0
3	:	2	=	1	Rest 1
1	:	2	=	0	Rest 1

Wir lesen von unten nach oben die Reste ab: 110_2

(c) 1010_2

(d) 1011_2

(e) 1111_2

(f)

27	:	2	=	13	Rest 1
13	:	2	=	6	Rest 1
6	:	2	=	3	Rest 0
3	:	2	=	1	Rest 1
1	:	2	=	0	Rest 1

Wir lesen von unten nach oben die Reste ab: 11011_2

(g) 101000_2

(h) 10000000_2

(i) 11111111_2

Lösung zu Aufgabe 13

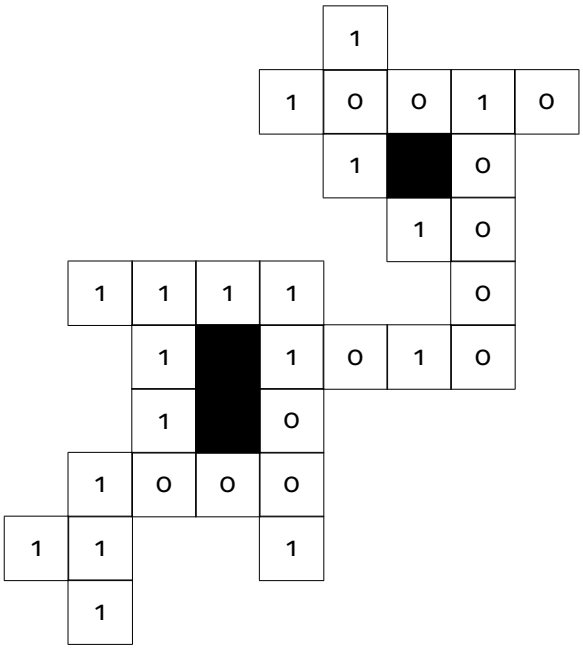
Wir müssen lediglich die letzte Ziffer anschauen. Ist diese eine 0, so ist die Zahl gerade, ansonsten ungerade. Die Begründung dafür ist einleuchtend: Die letzte Ziffer steht für die Einer, es ist die einzige Potenz (Wertigkeit), die ungerade ist. Mit den restlichen Stellen können wir nie ungerade Summen bilden.

Lösung zu Aufgabe 14

Eine Binärzahl können wir verdoppeln, indem wir eine 0 hinten anhängen. Die bisherigen Stellen sind nun alle doppelt so viel "wert" (sie stehen nun jeweils für die nächsthöhere Potenz), also ist auch ihre Summe doppelt so gross.

Im Zehnersystem bewirkt das Anhängen einer 0 eine Verzehnfachung der Zahl, jede Stelle hat dann einen 10 mal höheren Wert.

Lösung zu Aufgabe 17



Lösung zu Aufgabe 19

(a) `0x4`

(b) `0xF`

(c) `0x10`

(d) `0x7F`

(e) `0xFF`

Lösung zu Aufgabe 21

(a) 12

(b) 186 ($12 \cdot 16 + 10 \cdot 1$)

(c) 122 ($7 \cdot 16 + 10 \cdot 1$)

(d) 43962 ($10 \cdot 16^3 + 11 \cdot 16^2 + 11 \cdot 16^1 + 10 \cdot 16^0$)

(e) 64206

Lösung zu Aufgabe 24

(a) KLOSTER = 01010010110111010010100110010010001
SCHULE = 10010000100011110100010110010 0

(b) STIFT

(c) BINARYROCKS

Lösung zu Aufgabe 27

1001001 0100000 1101100 1101111 1110110 1100101 0100000 1111001 1101111 1110101 0100001

Die Leerzeichen dienen nur der besseren Lesbarkeit!

Lösung zu Aufgabe 28

The problem with ASCII lies in its beginning.

Lösung zu Aufgabe 31

(a) 11100010 10000010 10101100

(b) Der Codepoint für das Zeichen *ã* ist U+00E3

Lösung zu Aufgabe 32

Die Präfixe von ▬•▬ sind:

▬ (T)
▬• (N)
▬•• (D)

Lösung zu Aufgabe 34

Die Kodierungen (b) und (c) sind präfixfrei.
Bei (a) bildet das Code-Wort 0 den Anfang des Code-Wortes 01.
Bei (d) ist 10 ein Präfix von 100 und 1001. Zudem ist 100 ebenfalls ein Präfix von 1001.

Lösung zu Aufgabe 36

Die Umkehrung gilt nicht: Nicht alle eindeutigen Kodierungen sind präfixfrei! Ein Beispiel ist die Kodierung (d) in Aufgabe 34. Diese ist nicht präfixfrei, aber trotzdem eindeutig: Sie ist nämlich **suffixfrei**, und suffixfreie Kodierungen sind ebenfalls eindeutig. Die intuitive Begründung dahinter ist folgende: Wenn man in einer suffixfreien Kodierung alle Code-Wörter rückwärts anschauen, erhält man eine präfixfreie Kodierung, welche eindeutig ist. Ein Text, der mit einer suffixfreien Kodierung kodiert wurde, kann man also von hinten gelesen auf jeden Fall eindeutig dekodieren.

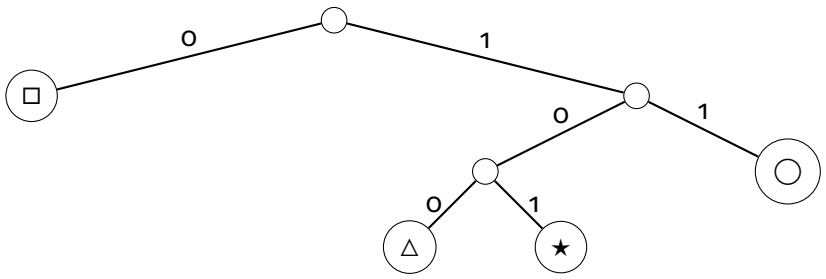
Es gibt aber auch Kodierungen, die weder präfix- noch suffixfrei und trotzdem eindeutig sind! Ein Beispiel dafür ist `{11, 101, 1011}`.

Lösung zu Aufgabe 38

(a) Bei einer Kodierung fixer Länge benötigen wir für 4 verschiedene Zeichen 2 Bits pro Zeichen. Beispiellösung:

□	00
◯	01
△	10
★	11

(b) *Eine* von vielen möglichen Lösungen ist der Codebaum aus dem Beispiel *Präfixfreie Kodierungen*, wobei wir die Zeichen A, B, C und D einfach durch □, ◯, △ und ★ ersetzen:



Wir weisen dabei denjenigen Zeichen, die häufiger vorkommen, kürzere Code-Wörter zu als denjenigen, die weniger häufig vorkommen.

(c) Die Nachricht □◯△□◯□◯□□★□△□◯□◯□ würde wie folgt kodiert:

Kodierung fixer Länge mit Beispiellösung aus (a):
`00011000010001000011001000010010 00` : 34 Bits

Kodierung mit Codebaum aus Beispiellösung für (b):
`01110001101100101010001101 10` : 28 Bits

(d) Wir sparen also 6 Bits.

(e) Die Nachricht □□□□ ◯ ◯ ◯ △△★ würde wir folgt kodiert:

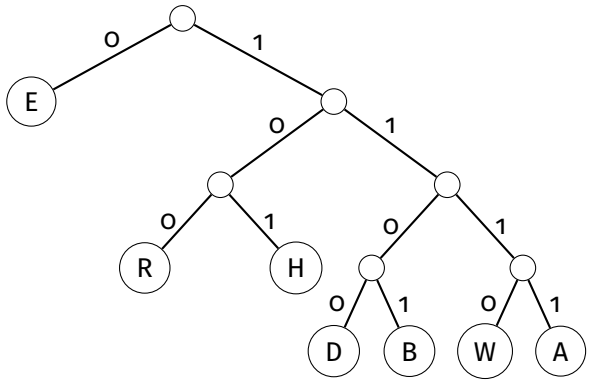
Kodierung fixer Länge mit Beispiellösung aus (a):
`00000000010101101011` : 20 Bits

Kodierung mit Codebaum aus Beispiellösung für (b):
`00001111111100100101` : 19 Bits

Wir sparen also nur 1 Bit in dieser Beispiellösung.

Lösung zu Aufgabe 39

Ein möglicher Huffman-Baum für das Wort ERDBEERWAEHE ist der folgende:



(a) Damit wird das Wort wie folgt kodiert: `01001100110100100111011110101O`

(b) Wir benötigen 30 Bits.

(c) Wir müssen 7 verschiedene Zeichen kodieren. Dafür benötigen wir 3 Bits pro Zeichen. Das Wort besteht aus 12 Buchstaben, würde also mit einer Kodierung fixer Länge mit $3 \cdot 12 = 36$ Bits kodiert

Lösung zu Aufgabe 41

$\frac{800}{40} = 20$ Pixel

Lösung zu Aufgabe 42

255 x 255 Pixel. 255 ist die Grösste Zahl, die wir binär mit 8 Bits darstellen können.

Lösung zu Aufgabe 43

Die Grafik hat eine Auflösung von 7x7 Pixel. Der Header muss also wie folgt aussehen:

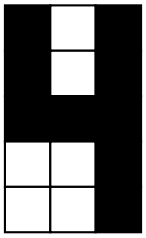
0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	1	1

Danach folgt der Payload, welcher aus 49 Bits besteht:

1	1	0	1	0	1	1	1	0	0	0	0	0	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0

1	0	0	0	0	0	1	1	0	1	0	1	0	1	1	0	1	0	1	0	1

Lösung zu Aufgabe 44



Lösung zu Aufgabe 46

(a) Mit 4 Bits können wir $2^4 = 16$ Graustufen darstellen.

(b) Schwarz: `0000` (0), weiss: `1111` (15)

(c) Damit können wir 2^5 verschiedene Abstufungen unterscheiden: 0 (schwarz) bis 31 (weiss), also `00000` bis `11111`

(d) Dafür brauchen wir 7 Bits.

Lösung zu Aufgabe 50

Wir brauchen pro Farbkanal 2 Bits, um 4 Zustände zu unterscheiden. Die Farbiete ist daher 6 Bit, da wir für jeden der drei Farbkanäle 2 Bits benötigen.

Wenn wir 32 Farben darstellen möchten, brauchen wir pro Kanal 5 Bits, total also 15 Bits.

Lösung zu Aufgabe 51

(a) $800 \cdot 600 \cdot 1$ Bit = 480000 Bits = 60000 Bytes = 60 KiloBytes

(b) $800 \cdot 600 \cdot 4$ Bits = 1920000 Bits = 240000 Bytes = 240 KiloBytes

(c) $800 \cdot 600 \cdot 12$ Bits = 5760000 Bits = 720000 Bytes = 720 KiloBytes

(d) $800 \cdot 600 \cdot 24$ Bits = 11520000 Bits = 1440000 Bytes = 1440 KiloBytes