Kantonsschule Im Lee

Informatik

Kolmogorov-Komplexität

Skript

Thomas Graf

Inhaltsverzeichnis

Kapitel 1

Kolmogorov-Komplexität

In der Theorie der algorithmischen Informationstheorie, ist die Kolmogorov[1]-Komplexität eines Objekts, wie zum Beispiel einem gegebenen Text, die Länge eines kürzesten Computerprogramms (in einer festgelegten Programmiersprache), welches dieses Objekt als Ausgabe (Output) produziert.

Die Kolmogorov-Komplexität gibt uns eine Möglichkeit den Informationsinhalt von Objekten zu bestimmen und erlaubt uns über kürzeste Darstellungen zu sprechen. Zudem erlaubt sie uns eine sinnvolle Definition der Bedeutung von Zufälligkeit von Texten und Zahlen. *Zufall* ist ein zentraler Begriff der modernen Wissenschaft.

1.1 Intuition

Betrachte die wunderschöne Illustration eines Teils der Mandelbrotmenge in Abbildung 1.1.

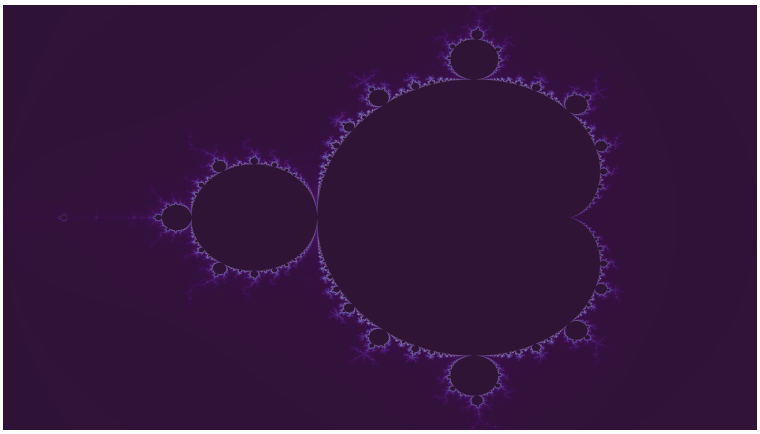

Abbildung 1.1: Mandelbrotmenge in 16k Auflösung

Wird die Illustration in Abbildung 1.1 Pixel für Pixel abgespeichert, benötigt die PNG-Kompression dieser Illustration rund 100 Millionen Bits an Speicherplatz, um die Illustration zu beschreiben. Nun kann aber ein recht kurzes Computerprogramm (siehe Programm A.1) diese 100 Millionen Bits reproduzieren, indem es von der Definition der Mandelbrotmenge Gebrauch macht. Dieses (kurze) Computerprogramm ist also eine deutlich kürzere Beschreibung der Illustration als die Angabe jedes Pixels. Für die Reproduktion dieser 100 Bits benötigt unser Computerprogramm zahlreiche

[1] Andrei Nikolajewitsch Kolmogorow, herausragender sowjetischer Mathematiker des 20. Jahrhunderts und Begründer der algorithmischen Komplexitätstheorie

Rechenoperationen und viel Rechenzeit. Dies ändert aber nichts daran, dass das Programm eine kurze Beschreibung dieser Illustration der Mandelbrotmenge ist. An dieser Stelle können wir wärmstens empfehlen, einen Blick in das Werk[1] zu werfen.

Aufgabe 1.1

Betrachte die folgenden zwei Bilder. Abbildung 1.2 zeigt ein Muster, erzeugt durch zahlreiche Kreise und Abbildung 1.3 zeigt zufälliges Rauschen (white noise).

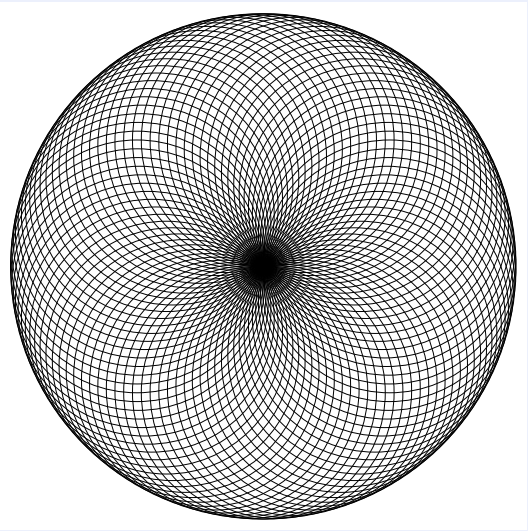

Abbildung 1.2: Kreise

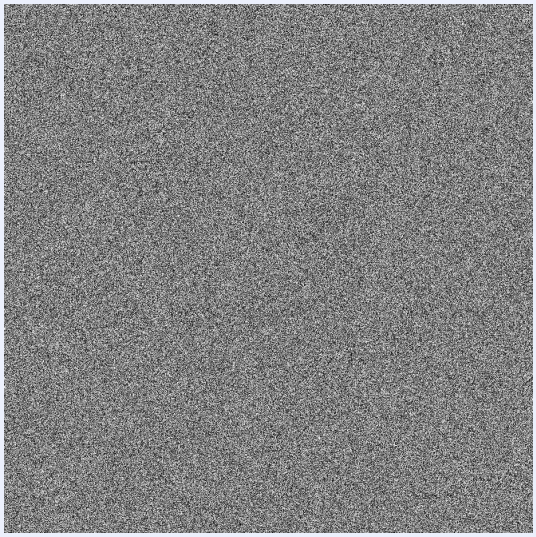

Abbildung 1.3: Rauschen

Welches dieser beiden Bilder erlaubt vermutlich eine kürzere Beschreibung? Begründe deine Antwort.

Lösungsvorschlag zu Aufgabe 1.1

Das Bild mit dem zufälligen Rauschen (Abbildung 1.3) erlaubt keine kurze Beschreibung. Mehr oder weniger muss jeder Pixel einzeln beschrieben werden. Das Muster der Kreise in Abbildung 1.2 ist sehr regelmässig und kann durch ein kurzes Computerprogramm beschrieben werden, das in einer Schleife Kreise an bestimmten Positionen zeichnet (wir erinnern uns an die Python-Turtle). Das zufällige Rauschen in Abbildung 1.3 hingegen weist keine erkennbare Struktur oder Regelmässigkeit auf. Das kürzeste Programm, das dieses spezifische Rauschbild erzeugt, muss wahrscheinlich die Farbinformation (weiss / schwarz) für jedes einzelne Pixel speichern. Die Beschreibung wäre also in etwa so gross wie das Bild selbst und damit viel länger als das Programm für das Kreismuster.

Bemerkung 1.1:
In diesem Kapitel fassen wir Wörter (Texte) als Träger von Information auf. Obwohl das Konzept der Kolmogorov-Komplexität auf beliebige Objekte anwendbar ist, werden wir uns hier der Einfachheit halber nur mit binären Wörtern befassen.

Wir werden nun allmählich versuchen eine robuste Methode zur Messung des Informationsgehalts eines Wortes zu entwickeln und dabei mehrfach auf Kapitel A verweisen.

Beispiel 1.1:
Alice hat ein langes binäres Passwort für den Zugriff auf eine Streaming-Platform gewählt. Bob möchte gerne die neueste Staffel seiner Lieblingsserie schauen und ruft dazu Alice an, um sie um das Passwort zu bitten[a].

- Angenommen Alices Passwort lautet $p_0 := 010110101100011011011111010$. Wie könnte sie Bob das Passwort p_0 per Telefon mitteilen? Ihr wird (mehr oder weniger) nichts anderes übrig bleiben, als das Passwort Bob Bit um Bit (Ziffer um Ziffer) zu diktieren. p_0 lässt sich also nicht kurz beschreiben und man kann sich p_0 auch nicht einfach merken.
- Anders hingegen wäre die Situation bei dem Passwort

$$p_1 := 1110011100111001110011100111001110011100 = (11100)^7,$$

Das Passwort p_1 weist eine starke Regelmässigkeit auf. Natürlich könnte Alice auch dieses Passwort Bit für Bit an Bob diktieren. Offensichtlich ist es jedoch einfacher, wenn sie ihm einfach mitteilt:

„Du erhältst das Passwort, indem du siebenmal das Wort 11100 hintereinander schreibst."

Dadurch hat Alice die Darstellung von p_1 *komprimiert*. Bitte beachte, dass Bob sich dieses Passwort p_1 recht einfach merken könnte.

[a] Bob scheint nicht mit dem *BitTorrent* Kommunikationsprotokoll (peer-to-peer file sharing) vertraut zu sein. Vielleicht benötigt er aber auch nur einen Vorwand, um Alice anzurufen.

Beispiel 1.1 motiviert die folgende intuitive Vorstellung: Wir wollen einem Wort einen kleinen Informationsgehalt beimessen, falls es eine kurze Darstellung / Beschreibung besitzt (komprimierbar ist) und einen grossen Informationsgehalt, falls das Wort so unregelmässig ist, dass keine kurze Darstellung / Beschreibung zulässt.

Definition 1.1 (Komprimierung intuitiv)**:**
Die Erstellung einer kürzeren Darstellung eines Wortes w nennen wir eine Komprimierung von w.

Aufgabe 1.2

Es sei S die Menge aller binären Wörter der Länge 9. Nun „ziehst" du zufällig und mit uniformen Wahrscheinlichkeitsverteilung ein Wort aus S.

1. Mit welcher Wahrscheinlichkeit wirst Du das Wort 010010110 ziehen?
2. Mit welcher Wahrscheinlichkeit wirst Du das Wort 111111111 ziehen?

Lösungsvorschlag zu Aufgabe 1.2

Die Menge S enthält genau $2^9 = 512$ verschiedene binäre Wörter der Länge 9.

1. Die Wahrscheinlichkeit, das Wort 010010110 zu ziehen, ist $\frac{1}{512}$.
2. Die Wahrscheinlichkeit, das Wort 111111111 zu ziehen, ist ebenfalls $\frac{1}{512}$.

Aufgabe 1.3

Intuitiv sollte *zufällig* bedeuten: „nach keinem klaren Plan gebaut“. Ein zufälliges Objekt weist also eine chaotische (und keine regelmässige) Struktur auf. Vergleiche mit Aufgabe 1.2 und argumentiere, warum die klassische Wahrscheinlichkeitsrechnung diese Intuition nicht einfängt.

Lösungsvorschlag zu Aufgabe 1.3

Wie in Aufgabe 1.2 gesehen, ist aus Sicht der klassischen Wahrscheinlichkeitstheorie jedes binäre Wort der Länge 9 gleich wahrscheinlich. Die klassische Wahrscheinlichkeitstheorie macht also keine Aussage über die Zufälligkeit eines einzelnen Wortes, sondern nur über die Wahrscheinlichkeit, mit der ein Wort aus einer Menge von Wörtern gezogen wird.

Obwohl das Wort 111111111 sehr regelmässig und das Wort 010010110 sehr unregelmässig erscheint, sind beide aus Sicht der klassischen Wahrscheinlichkeitstheorie gleich wahrscheinlich. Die klassische Wahrscheinlichkeitstheorie kann also nicht zwischen der Zufälligkeit einzelner Objekte unterscheiden. Genau diese Lücke wird durch die Kolmogorov-Komplexität geschlossen.

Ein denkbares Vorgehen zur Messung des Informationsgehalts eines binären Worts w wäre, sich auf eine konkrete Komprimierungsmethode *komp* zu einigen. Die Länge $|komp(w)|$ des komprimierten Wortes $komp(w)$ könnte dann als Mass für den Informationsgehalt von w angesehen werden.

Natürlich muss auch $komp(w)$ ein Wort über dem binären Alphabet sein. Würden wir $komp(w)$ in einem anderen Alphabet darstellen, wäre der Vergleich der Darstellungslängen nicht mehr sinnvoll, denn schliesslich ist es immer möglich, ein Wort in einem mächtigen Alphabet kurz darzustellen.

1.2 Problematik einer fest gewählten Komprimierung

Es existieren unendlich viele Komprimierungsmethoden, welche für eine feste Wahl einer Komprimierung zur Verfügung stehen. Doch welche Komprimierungsmethode stellt nun die korrekte Wahl dar? Das Problem wird sein, dass, egal welche Komprimierung verwendet wird, immer eine andere Komprimierung existiert, welche für unendlich viele Wörter kürzere Darstellungen erzeugt.

Beispiel 1.2 (Komprimierung durch Ausnutzung von Wiederholungen)**:**
Lass uns ein binäres Wort mit zahlreichen Wiederholungen von Teilwörtern untersuchen. Konkret wollen wir das Wort

$$w := 00(101)^{25}(01010)^{9}(1011)^{16} = 00(101)11001(01010)1001(1011)10000$$

untersuchen, wobei wir die Exponenten in Basis 10 durch ihre binäre Darstellung ersetzt haben. Die resultierende Darstellung ist jedoch noch immer kein binäres Wort. Tatsächlich ist es ein Wort über dem Alphabet $\{0, 1, (,)\}$, welches vier Symbole erhält. Mit der Vereinbarung

$$0 \to 00, \quad 1 \to 11, \quad (\to 10, \quad) \to 01$$

erhalten wir die doppelt so lange (aber dafür binäre) Darstellung:

$$w = 000010110011011111000011100011001100011100001110110011110111000000000.$$

Beispiel 1.3 (Komprimierung durch Primfaktorisierung)**:**
Es sei x ein binäres Wort. Falls $dec(x) \geq 2$, dann gilt

$$dec(x) = p_1^{e_1} p_2^{e_2} \dots p_t^{e_t},$$

wobei die p_i unterschiedliche Primzahlen sind und $e_i \geq 0$ für $i \in \{\, n \in \mathbb{N} \,;\, 1 \leq n \leq t \,\}$. Eine denkbare Darstellung von $p_1^{e_1} p_2^{e_2} \dots p_t^{e_t}$ ist

$$bin(p_1)(bin(e_1))bin(p_2)(bin(e_2)) \dots bin(p_t)(bin(e_t)).$$

Mit der Vereinbarung aus Beispiel 1.2 erhalten wir wieder eine binäre Darstellung von x.

Leider sind die beiden Komprimierungsmethoden aus Beispiele 1.2 und 1.3 nicht miteinander vergleichbar. Die erste Methode (Ausnutzung von Wiederholungen) erzeugt für gewisse Wörter eine kürzere Darstellung als die zweite Methode (Primfaktorisierung) und umgekehrt.

Beispiel 1.4 (Unvergleichbarkeit von Komprimierungsmethoden)**:**

- Für das Wort $w_A := (10110010)^{512}$ mit der binären Länge von $8 \cdot 512 = 4096$ Bits, ergibt die Komprimierung aus Beispiel 1.2:

$$\begin{aligned} w_A = (10110010)^{512} = (10110010)bin(512) = (10110010)1000000000 = \\ = \underbrace{10}_{(}\underbrace{1100111100001100}_{10110010}\underbrace{01}_{)}\underbrace{1100000000000000000}_{1000000000} \end{aligned}$$

  also als zusammenhängendes binäre Wort:

$$1010110011110000110001011100000000000000$$

  mit einer Darstellungslänge von 40 Bits gegenüber der unkomprimierten Darstellungslänge von 4096 Bits.
- Für das Wort $w_B := bin\left(3^{5000} \cdot 5^{4000} \cdot 17^{3000}\right)$ mit der binären Länge von 17230 Bits (siehe Theorem A.3), ergibt die Komprimierung aus Beispiel 1.3:

$$\begin{aligned} w_B &= bin\left(3^{5000} \cdot 5^{4000} \cdot 17^{3000}\right) = \\ &= bin(3)(bin(5000))bin(5)(bin(4000))bin(17)(bin(3000)) = \\ &= 11(1001110001000)101(111110100000)10001(101110111000). \end{aligned}$$

  Nach Anwendung unserer Vereinbarung

$$0 \to 00, \quad 1 \to 11, \quad (\to 10, \quad) \to 01$$

  erhalten wir eine Darstellungslänge von 106 Bits gegenüber der unkomprimierten Darstellungslänge von 17230 Bits.
- Die Komprimierungsmethode der Primfaktorisierung für w_A wird vermutlich kaum gewinnbringend sein, da $dec(w_A)$ vermutlich nicht als Produkt hoher Potenzen weniger Primzahlen geschrieben werden kann. Umgekehrt wird w_B erwartungsweise keine Muster von Wiederholungen von Nullen und Einsen aufweisen, deren Ausnützung sich lohnen würde.

Somit sind die beiden Komprimierungsmethoden aus den Beispielen 1.2 und 1.3 nicht miteinander vergleichbar. Die erste Methode (Ausnutzung von Wiederholungen) erzeugt für gewisse Wörter eine deutlich kürzere Darstellung als die zweite Methode (Primfaktorisierung) und umgekehrt.

1.3 Idee hinter der Kolmogorov-Komplexität

Die Definition eines Komplexitätsmasses sollte robust sein in dem Sinne, dass sie nicht auf einer arbiträren Wahl einer Komprimierungsmethode basiert. Ein Komplexitätsmass, welches keine arbiträre Wahl verwendet, ist die Kolmogorov-Komplexität. Um deren Definition zu verstehen, müssen wir zunächst den Begriff der *Generierung* eines Worts einführen.

Definition 1.2 (Generierung eines Worts)**:**
Es sei w ein beliebiges Wort. Wir sagen, dass ein Programm (ein Algorithmus) A das Wort w generiert, falls der Aufruf $A()$ genau das Wort w ausgibt.

Beispiel 1.5:
Das Programm

```
#include <iostream>
void A() {
    std::cout << "00110111" << std::endl;
}
```

generiert das Wort 00110111.

Beispiel 1.6:
Das Programm

```
#include <iostream>
#include <string>
void B() {
    std::string wort(9900507, '1');
    std::cout << wort << std::endl;
}
```

generiert das Wort $w := 1^{9900507} = \underbrace{11\ldots111}_{9900507 \text{ Einsen}}$.

Beachte, dass der Algorithmus `B` eine kurze Beschreibung von w relativ zur grossen Länge von w darstellt.

Die Idee von Kolmogorov besteht darin, dass wir uns nicht auf eine arbiträre Komprimierung einigen müssen. Stattdessen erlaubt er, beliebige Programme (in einer fest gewählten Programmiersprache) zur Beschreibung von Texten. Wenn ein Text durch ein kurzes Programm beschrieben werden kann, so hat es eine geringe Kolmogorov-Komplexität. Erlaubt ein Text hingegen keine kurze Beschreibung als Programm, so besitzt er eine hohe Kolmogorov-Komplexität.

1.4 Selbstbegrenzung von Programmen und Definition der Kolmogorov-Komplexität

Zunächst möchten wir das kleine, aber mühsame Detail der sogenannten *Selbstbegrenzung* adressieren und die damit verbundene Problematik in diesem Abschnitt aus dem Weg räumen. Betrachten wir dazu nochmals das folgende (standard) `C++`-Programm aus Beispiel 1.5:

```
#include <iostream>
void A() {
    std::cout << "00110111" << std::endl;
}
```

Das Problem ist, dass wir typischerweise die Symbole der Tastatur in Binärcode durch Folgen von 7 Nullen und Einsen (zum Beispiels mittels der ASCII-Kodierung) darstellen müssen (natürlich gibt es auch noch andere Möglichkeiten). Wenn wir aber das gesamte Programm `A` so kodieren, dann würden wir für das Wort $w := 00110111$ nicht nur $|w|$ Bits benötigen, sondern $7\,|w|$. Diesen Faktor 7 möchten wir gerne vermeiden. Damit unser Programm genau das Wort w (weder mehr noch weniger) ausgibt, muss klar sein, welcher Teil im Programm `A` dem Wort w entspricht und welcher Teil dem Rest des Programms. Um dies zu erreichen, designen wir eine ganz leicht modifizierte Version des `g++` Compilers und der Programmiersprache `C++`. Lass uns diesen modifizierten Compiler `g'++` nennen und die angepasste Sprache `C'++`.

Die Sprache `C'++` unterscheidet sich von standard `C++` lediglich darin, dass wir in einem `C'++` Programm die Escape-Sequenzen `***` und `+++` verwenden dürfen. Der modifizierte Compiler `g'++` übersetzt dann das Programm ganz normal in Maschinencode, integriert aber die **binäre Darstellung** der Programmteile, deren Beginn mit `***` und Ende mit `+++` gekennzeichnet wird, direkt in den Maschinencode. Eine Verwendung dieser Escape-Sequenzen für einen anderen Zweck wäre dann ein semantischer Fehler.

Beispiel 1.7:
Das Programm

```
#include <iostream>
void A() {
    std::cout << "***00110111+++" << std::endl;
}
```

generiert das Wort $w := 00110111$ und ist eine gültiges Programm in Sprache `C'++`. Der Programmteil 00110111 wird bei der Kompilierung des Programms mithilfe von `g'++` unverändert in dem Maschinencode vorkommen, wird also exakt 8 Bits zu Länge des Maschinencodes beitragen.

Nun können wir endlich die formale Definition der Kolmogorov-Komplexität angeben:

Definition 1.3 (Kolmogorov-Komplexität)**:**
Es sei w ein binäres Wort. Die Kolmogorov-Komplexität $K(w)$ von w ist definiert als das Minimum der **binären Längen** aller `C'++` Programme (kompiliert mit `g'++`), die w generieren.

Für ein binäres Wort w betrachten wir alle (unendlich vielen) Maschinencodes der `C'++` Programme, die w generieren. Die Länge eine kürzesten[2] solcher Maschinencodes ist dann die Zahl $K(w)$.

[2]Im Allgemeinen kann es mehrere verschiedene kürzeste Maschinencodes geben, die w generieren.

Bemerkung 1.2:
Ist $K(w)$ ein geeignetes Mass für den Informationsgehalt des binären Wortes w? Ja, da jede Komprimierungsmethode als Programm formuliert werden kann, bezieht die Kolmogorov-Komplexität jede denkbare Komprimierungsmethode ein. Es sei x ein binäres Wort und f eine Komprimierungsmethode (Funktion), die zu x eine komprimierte Darstellung $x' := f(x)$ generiert. Es sei f^{-1} die Umkehrung der Komprimierung f, also $f^{-1}(f(x)) = f^{-1}(x') = x$. Dann können wir ein Programm schreiben, welches lediglich (das kurze Wort) x' als Parameter beinhaltet und x wieder aus x' mithilfe von f^{-1} erzeugt und schliesslich ausgibt. Dazu muss f^{-1} natürlich auch im Programm kodiert sein. Doch die binäre Länge der Beschreibung dieser Umkehrfunktion ist unabhängig von x und x' und kann somit als konstant angesehen werden. Das so aufgebaute Programm würde dann x generieren, ohne die Darstellung von x speichern zu müssen.

In Abschnitt 1.5 werden wir sehen, dass die Wahl der Programmiersprache keine wesentliche Rolle spielt.

1.5 Invarianz der Programmiersprache

Man könnte sich denken, dass die Wahl einer festen Programmiersprache wiederum ein arbiträres Element einbringt. Dem ist aber nicht so! Wir klären diesen Sachverhalt in Theorem 1.1.

Theorem 1.1 (Sprachinvarianz)**:**
Es seien A und B beliebige Programmiersprachen und w ein binäres Wort. Wir bezeichnen mit $K_S(w)$ die Länge des kürzesten Maschinencodes eines Programms in Sprache S, das w generiert. Es existiert eine Konstante $c_{A,B}$, welche nur von A und B abhängt, sodass

$$|K_A(w) - K_B(w)| \leq c_{A,B}$$

für alle binären Wörter w.

Beweis 1.1:

- Da die Sprachen A und B turingvollständig sind, existiert ein Programm $T_{B\to A}$ in Sprache A, welches jedes Programm in Sprach B in ein äquivalentes Programm in Sprache A übersetzt. Analog existiert ein Programm $T_{A\to B}$ in Sprache B, welches jedes Programm in Sprache A in äquivalentes Programm in Sprache B übersetzt.
- Es sei B_w ein Programm in Sprache B, welches w generiert. Dann können wir B_w dem Programm $T_{B\to A}$ als Parameter übergeben. Dann ist $T_{B\to A}(B_w)$ ein Programm, in Sprache A, welches w generiert. Wir bezeichnen die binäre Länge des Programms $T_{B\to A}$ mit $c_{B\to A}$. Somit gilt $K_A(w) \leq K_B(w) + c_{B\to A}$.
- Analog (von A zu B) finden wir $K_B(w) \leq K_A(w) + c_{A\to B}$.
- Der Unterschied $|K_A(w) - K_B(w)|$ ist also nicht grösser als

$$c_{A,B} := \max\{c_{A\to B}, c_{B\to A}\}.$$

Theorem 1.1 besagt also, dass die konkrete Wahl der Programmiersprache für die Definition der Kolmogorov-Komplexität keine wesentliche Rolle spielt.

1.6 Triviale obere Schranke für die Kolmogorov-Komplexität

Ein beliebiges binäres Wort w kann sicherlich immer beschrieben werden, indem man jedes Bit des Wortes einzeln angibt, w also Bit für Bit diktiert. Diese Beobachtung lässt vermuten, dass die Kolmogorov-Komplexität eines jeden binären Wortes w zumindest nicht (wesentlich) länger ist, als die Anzahl Bits von w (Länge von w). Diese Vermutung ist korrekt, wie Theorem 1.2 zeigt.

Theorem 1.2 ($K(w)$ ist auf keinen Fall wesentlich länger als $|w|$)**:**
Es existiert eine Konstante c, sodass für jedes binäre Wort w gilt

$$K(w) \leq |w| + c.$$

Beweis 1.2:
Es genügt, ein Programm anzugeben, welches ein beliebiges binäres Wort w generiert und eine binäre Länge aufweist, welche kleiner oder gleich $|w| + c$ ist (für eine geeignete Konstante c). Das Programm

```
#include <iostream>
void A() {
    std::cout << "***w+++" << std::endl;
}
```

generiert w. Der gelbe Teil des Programms ist identisch für jedes Wort w. Das Wort `w` wird unverändert (binär) im Maschinencode vorkommen. Der resultierende Maschinencode hat also eine Länge von $|w|$ vielen Bits (für die Darstellung des binären Wortes w) und zusätzlich noch irgendeine konstante Anzahl Bits c für die Kodierung des eigentlichen Programms (gelber Teil). Die Anzahl Bits c ist konstant, in dem Sinne, dass sie nicht von w abhängig ist. Damit haben wir ein Programm angegeben, welches w generiert und eine binäre Länge von $|w| + c$ hat. Somit gilt $K(w) \leq |w| + c$ für eine Konstante c.

Die Kolmogorov-Komplexität eines Wortes w ist also sicherlich (bis auf eine Konstante) nicht grösser als die Länge von w.

1.7 Sehr regelmässige Wörter

Theorem 1.2 gibt lediglich eine obere Schranke für die Kolmogorov-Komplexität von binären Wörtern an. Wörter w, die eine hohe Regelmässigkeit aufweisen, müssen nicht Bit für Bit angegeben werden, sondern lassen sich (unter Ausnutzung ihrer Regelmässigkeit) deutlich kürzer beschreiben als mit $|w| + c$ vielen Bits.

Beispiel 1.8:
Es sei $w_n := 1^n$ für eine beliebige natürliche Zahl $n > 0$. Das Wort w_n besteht also genau aus n-vielen Einsen (und hat insbesondere die Länge n). Es handelt sich also um ein sehr regelmässiges Wort, welches (intuitiv) nicht Bit für Bit beschrieben werden muss. Das folgende Programm `Cn` generiert das Wort w_n:

```
#include <iostream>
#include <string>

void Cn() {
    std::string wort(***n+++, '1');
```

```
    std::cout << wort << std::endl;
}
```

Im Programm `Cn` ist einzig die binäre Kodierung `n` der Zahl n abhängig von n beziehungsweise von w_n. Der Rest des Programms ist unabhängig von der konkreten Wahl von n (für jede Wahl von n gleich). Gemäss Theorem A.3 benötigen wir zur binären Darstellung von n genau $\lceil \log_2(n+1) \rceil$ viele Bits. Damit existiert Konstanten c_0 und c_1, sodass

$$K(w_n) \leq c_0 + \lceil \log_2(n+1) \rceil \overset{\textit{Theorem A.4}}{\leq} \log_2(n) + c_1 = \log_2(|w_n|) + c_1$$

für beliebiges natürliches $n > 0$.

Beispiel 1.9:
Es sei $v_n := 1^{(n^2)}$ für eine beliebige natürliche Zahl $n > 0$. Das Wort v_n besteht also genau aus n^2-vielen Einsen. Es gilt $|v_n| = n^2$ und somit $n = \sqrt{|v_n|}$. Das folgende Programm `Dn` generiert das Wort v_n:

```
#include <iostream>
#include <string>

void Dn() {
    std::string wort;
    int M = ***n+++
    int M = M * M
    wort.reserve(M);

    for (int i = 0; i < M; ++i) {
        wort += '1';
    }

    std::cout << wort << std::endl;
}
```

Analog zu Beispiel 1.8, ist auch hier einzig die binäre Kodierung `n` der Zahl n abhängig von n beziehungsweise von v_n. Der Rest des Programms ist wieder unabhängig von der konkreten Wahl von n (für jede Wahl von n gleich). Zur binären Darstellung von n benötigen wir genau $\lceil \log_2(n+1) \rceil$ viele Bits. Damit existiert Konstanten c_2 und c_3, sodass

$$K(v_n) \leq c_3 + \lceil \log_2(n+1) \rceil \overset{\textit{Theorem A.4}}{\leq} \log_2(n) + c_4 = \log_2\left(\sqrt{|v_n|}\right) + c_4$$

für beliebiges natürliches $n > 0$.

Bei der Bestimmung einer oberen Schranke für die Kolmogorov-Komplexität eines Wortes, interessieren wir uns lediglich dafür, wie viele Bits wir für die **Beschreibung** dieses Wortes benötigen! Dass

- die Inhalte von Variablen wie zum Beispiel von `M` während er Ausführung des Programms `Dn` möglicherweise gigantisch gross werden,
- die Programmausführung sehr viel Speicher belegen könnte

- oder eine Berechnung wie `M * M` potenziell sehr viel Rechenzeit in Anspruch nehmen könnte

tut hier nichts zur Sache! Uns interessiert einzig und alleine die binäre Länge der **Beschreibung** eines Wortes. Wir gehen stets davon aus, dass die verwendeten Zahlentypen (wie `int`) die berechneten Werte korrekt abspeichern können (die Zahlenbereiche ausreichend gross sind) und genügend Speicher vorhanden ist.

Aufgabe 1.4

Betrachte nochmals Beispiel 1.9. Natürlich könnte das Wort v_n auch durch das folgende Programm beschrieben werden:

```
#include <iostream>
#include <string>

void Fn() {
    std::string wort;
    wort.reserve(***n**2+++);

    for (int i = 0; i < ***n**2+++; ++i) {
        wort += '1';
    }

    std::cout << wort << std::endl;
}
```

Warum beweist dieses Programm sicherlich eine schlechtere obere Schranke für die Kolmogorov-Komplexität von v_n als Beispiel 1.9?

Lösungsvorschlag zu Aufgabe 1.4

Da `Fn` zweimal die binäre Kodierung von n^2 enthält, kann `Fn` sicherlich keine bessere obere Schranke liefern als

$$K(v_n) \leq c + 2\left\lceil \log_2\left(n^2+1\right)\right\rceil \leq c' + 2\log_2\left(n^2\right) = c' + 4\log_2\left(\sqrt{|v_n|}\right)$$

für geeignete Konstanten c, c'.

1.8 Nichtkomprimierbare Wörter und Zufall

Wir haben die Kolmogorov-Komplexität nur für binäre Wörter definiert. Wir können die Kolmogorov-Komplexität aber auch ganz einfach für natürliche Zahlen definieren, indem wir die natürliche Zahl einfach in Basis 2 (binär) darstellen, um ein binäres Wort zu erhalten.

Definition 1.4 (Kolmogorov-Komplexität einer natürlichen Zahl)**:**
Die Kolmogorov-Komplexität $K(n)$ einer natürlichen Zahl n ist $K(n) := K(bin(n))$.

Es existieren Wörter, die nicht komprimierbar sind, die also eine Kolmogorov-Komplexität besitzen, die nicht kleiner ist als ihre Länge.

Theorem 1.3 (Existenz nichtkomprimierbarer Wörter)**:**
Für jede natürliche Zahl $n > 0$ existiert (mindestens) ein binäres Wort w der Länge n, für welches gilt

$$K(w) \geq n = |w|\,,$$

das heisst, für jede Länge n existiert (mindestens) ein Wort dieser Länge, welches sich nicht komprimieren lässt.

Insbesondere ist dadurch also für jedes natürliche $n > 0$ die Existenz (mindestens) eines binären Wortes w (wir sagen nichts über die Länge von w aus) mit $K(w) \geq n$ gesichert.

Beweis 1.3:
Unser Beweis verwendet ein einfaches kombinatorisches Argument (Abzählen): Ein Maschinencode kann höchstens ein binäres Wort generieren. Beachte auch, dass ein Maschinencode der Länge 0 (leerer Maschinencode) kein Wort generieren kann. Wie viele nichtleere Maschinencodes der Länge kleiner als n gibt es höchstens? Sicherlich sind es nicht mehr als es Elemente in der Menge

$$S := \{\ w\ ;\ w \text{ ist ein binäres Wort mit } 1 \leq |w| \leq n-1\ \}$$

gibt. Schliesslich ist jeder nichtleere Maschinencode insbesondere auch ein nichtleeres binäres Wort. Es gibt genau 2^1 binäre Wörter der Länge 1, 2^2 binäre Wörter der Länge 2 und allgemein 2^k binäre Wörter der Länge k. Insgesamt enthält die Menge S also genau

$$\sum_{k=1}^{n-1} 2^k \overset{\textit{Theorem A.2}}{=} 2^n - 2$$

Elemente. Doch es gibt genau 2^n binäre Wörter der Länge n. Dies sind aber mehr als es nichtleere verschiedene Maschinencodes der Länge $< n$ gibt. Damit muss ein binäres Wort x der Länge n existieren, für welches kein Maschinencode mit Länge $< n$ existiert und somit $K(x) \geq n = |w|$.

Ein nichtkomprimierbares Wort erlaubt also keine kürzere Beschreibung, als das Wort vollständig zu beschreiben. Es gibt also keinen anderen Plan zu seiner Generierung, als einzig seine vollständige Beschreibung. Damit ist es plausible, dass Definition 1.5 unsere bislang beste bekannte Formalisierung des informellen Begriffes „zufällig“ ist.

Definition 1.5 (Zufälliges binäres Wort, zufällige Zahl)**:**

- Ein binäres Wort w heisst *zufällig*, falls $K(w) \geq |w|$.
- Eine natürliche Zahl $n > 0$ heisst zufällig, falls $K(n) = K(bin(n)) \geq \lceil \log_2(n+1) \rceil - 1$.

Aufgabe 1.5

Woher kommt der Term -1 in der Definition einer zufälligen Zahl (siehe Definition 1.5)?

Lösungsvorschlag zu Aufgabe 1.5

Die kürzeste binäre Darstellung $bin(n)$ jeder natürlichen Zahlen $n > 0$ beginnt mit einer 1. Sonst hätte die Darstellung führende Nullen und es wäre nicht die kürzeste Darstellung. Das erste Bit in der binären Darstellung von n ist also bereits bekannt. Es ist also sinnvoll die Zahl auch schon als zufällig anzuschauen, falls alle Bits ausser dem ersten genannt werden müssen.

1.9 Die Kolmogorov-Komplexität ist nicht berechnenbar

Theorem 1.4 ($K(w)$ ist nicht für jedes binäre Wort berechenbar)**:**
Das Problem, für jedes binäre Wort w die Kolmogorov-Komplexität $K(w)$ zu berechnen, ist algorithmisch unlösbar (nicht berechenbar).

Mit anderen Worten: Es existiert kein Algorithmus, welcher ein beliebiges binäres Wort w als Eingabe erhält und die natürliche Zahl $K(w)$ als Ausgabe liefert.

Beweis 1.4:
Es sei $n > 0$ eine natürliche Zahl. Nun bezeichnen wir mit α das erste Wort bezüglich der kanonischen Ordnung über dem binären Alphabet, welches eine Kolmogorov-Komplexität von mindestens n hat. Wegen Theorem 1.3 und der Verwendung der kanonischen Ordnung ist die Existenz und Eindeutigkeit von α gesichert. Bitte beachten Sie, dass wir keine Aussage zur Länge von α machen. Insbesondere wird also nicht behauptet, dass α die Länge n hat.

Angenommen es existiert ein Algorithmus Q, der zu jedem binären Wort w die Zahl $K(w)$ berechnet. Dann können wir Q verwenden, um eine kurze Beschreibung von α anzugeben. Für jede natürliche Zahl $n > 0$ sucht und generiert das folgende `C'++` Programm[a] das Wort α:

```
#include <iostream>
#include <string>
void seeking_alpha_n() {
    x = leeres Wort;
    Kx = Q(x);
    while (Kx < ***n+++) {
        x = Nachfolger von x in der kanonischen Ordnung über {0, 1}^*;
        Kx = Q(x)
    }
    std::cout << x << std::endl;
}
```

Alle Algorithmen `seeking_alpha_n` sind identisch bis auf die Zahl n. Es sei c die Länge des Maschinencodes von `seeking_alpha_n` bis auf die Angabe von n. Damit ist bewiesen, dass $K(\alpha) \leq c + \lceil \log_2(n+1) \rceil$.

Nach Definition von α gilt aber auch $K(\alpha) \geq n$ für alle $n \in \mathbb{N}$ mit $n > 0$. Doch die Ungleichungen

$$n \leq K(\alpha) \leq c + \lceil \log_2(n+1) \rceil$$

können höchstens für endlich viele natürliche Zahlen $n > 0$ gelten. Dies ist ein Widerspruch zur Annahme, dass ein Algorithmus Q zur Berechnung der Kolmogorov-Komplexität von allen binären Wörtern existiert.

[a] genauer: in `C'++`-Pseudocode

1.10 Elemente rekursiver Sprachen haben eine tiefe Kolmogorov-Komplexität

TODO

1.11 Verständnisaufgaben

Aufgabe 1.6

Wie viele Stellen hat die kürzeste Darstellung der Zahl $n = 7^4 + 358$ in Basis $b = 6$?

Lösungsvorschlag zu Aufgabe 1.6

Die gesuchte Länge ist

$$\left\lceil \log_6 \left(7^4 + 358 + 1\right) \right\rceil = \left\lceil \ln \left(7^4 + 359\right) / \ln (6) \right\rceil = 5.$$

Aufgabe 1.7

Berechne

$$\sum_{k=0}^{9} \left(5 \cdot 4^k\right).$$

Lösungsvorschlag zu Aufgabe 1.7

$$\sum_{k=0}^{9} \left(5 \cdot 4^k\right) = 5 \cdot \sum_{k=0}^{9} \left(4^k\right) = 5 \cdot \frac{4^{10} - 1}{4 - 1} = \\ = 1747625$$

Aufgabe 1.8

Es sei $q \neq 0$ eine reelle Zahl und n und $m \leq n$ natürliche Zahlen. Dann gilt

$$\sum_{k=m}^{n} q^k = q^m + q^{m+1} + \ldots + q^n = \frac{q^{n+1} - q^m}{q - 1}.$$

Lösungsvorschlag zu Aufgabe 1.8

Siehe den Beweis von Theorem A.2.

Aufgabe 1.9

Wie viele Elemente enthält die Menge

$$S := \{\, w \;;\; w \text{ ist ein binäres Wort mit } 1 \leq |w| \leq n-1 \,\}$$

Lösungsvorschlag zu Aufgabe 1.9

Es gibt genau 2^1 binäre Wörter der Länge 1, 2^2 binäre Wörter der Länge 2 und allgemein 2^k binäre Wörter der Länge k. Insgesamt enthält die Menge S also genau

$$\sum_{k=1}^{n-1} 2^k \overset{\textit{Theorem A.2}}{=} 2^n - 2$$

Elemente.

Anhang A

Details

A.1 Werkzeuge

Definition A.1 (*bin* und *dec*)**:**

- Es sei $n > 0$ eine natürliche Zahl. Dann bezeichnen wir mit $bin(n)$ die binäre (Basis 2) Zahlendarstellung von n ohne führende Nullen.
- Es sei $n > 0$ eine natürliche Zahl. Dann bezeichnen wir mit $dec(n)$ die dezimale (Basis 10) Zahlendarstellung von n ohne führende Nullen.
- Wir definieren zusätzlich $bin(0) := 0$ sowie $dec(0) := 0$.

Theorem A.1 (geometrische Summe)**:**
Es sei $q \neq 0$ eine reelle Zahl und n eine natürliche Zahl. Dann gilt

$$\sum_{k=0}^{n} q^k = q^0 + q^1 + \ldots + q^n = \frac{q^{n+1} - 1}{q - 1}.$$

Beweis A.1:
Wir beweisen die Aussage durch vollständige Induktion.

- Für $n = 0$ gilt die Aussage, da

$$\sum_{k=0}^{0} q^k = q^0 = 1 = \frac{q^{0+1} - 1}{q - 1}.$$

- Die Aussage gelte nun für eine natürliche Zahl n. Wir zeigen, dass sie auch für $n+1$ gilt.

$$\sum_{k=0}^{n+1} q^k = q^{n+1} + \sum_{k=0}^{n} q^k = q^{n+1} + \frac{q^{n+1} - 1}{q - 1} = \frac{q^{n+2} - 1}{q - 1}.$$

Theorem A.2 (verallgemeinerte geometrische Summe)**:**

Es sei $q \neq 0$ eine reelle Zahl und n und $m \leq n$ natürliche Zahlen. Dann gilt

$$\sum_{k=m}^{n} q^k = q^m + q^{m+1} + \ldots + q^n = \frac{q^{n+1} - q^m}{q-1}.$$

Beweis A.2:
Unter Verwendung von Theorem A.1 finden wir

$$\sum_{k=m}^{n} q^k = \sum_{k=0}^{n} q^k - \sum_{k=0}^{m-1} q^k = \frac{q^{n+1}-1}{q-1} - \frac{q^m - 1}{q-1} = \frac{q^{n+1} - q^m}{q-1}.$$

Theorem A.3 (Anzahl Ziffern in Zahlendarstellung)**:**
Es seien $n > 0$ und $b > 1$ natürliche Zahlen. Die kürzeste b-adische Darstellung von n (Darstellung ohne führende Nullen) hat genau $\lceil \log_b(n+1) \rceil$ Stellen.

Beweis A.3:
Es sei s die Anzahl der Stellen der kürzesten b-adischen Darstellung von n. Die grösste s-stellige Zahl in Basis b (kürzeste Darstellung) ist

$$\sum_{k=0}^{s-1} (b-1)b^k = (b-1)\sum_{k=0}^{s-1} b^k = (b-1)\frac{b^s - 1}{b-1} = b^s - 1,$$

wobei wir Theorem A.1 verwendet haben. Die kleinste s-stellige Zahl Basis b (kürzeste Darstellung) ist b^{s-1}. Somit gilt

$$\begin{aligned} b^{s-1} - 1 < n \leq b^s - 1 &\iff \\ b^{s-1} < n + 1 \leq b^s &\iff \\ s - 1 < \log_b(n+1) \leq s, & \end{aligned}$$

wobei wir verwendet haben, dass $\log_b$ eine streng monoton wachsende Funktion ist. Dann folgt aber $\lceil \log_b(n+1) \rceil = s$.

Theorem A.4 (Obere Schranke binäre Darstellungslänge)**:**
Für jede natürliche Zahl $n > 0$ gilt

$$\lceil \log_2(n+1) \rceil \leq \lceil \log_2(n) \rceil + 1.$$

Offensichtlich folgt aus dieser Behauptung sofort

$$\lceil \log_2(n+1) \rceil \leq \lceil \log_2(n) \rceil + 1 < \log_2(n) + 2.$$

Beweis A.4:
Wir beweisen zunächst die Ungleichung $\log_2(n+1) \leq \log_2(n) + 1$. Dazu berechnen wir zunächst

$$\log_2(n) + 1 = \log_2(n) + \log_2(2) = \log_2(2n).$$

Damit gilt also

$$\begin{aligned} \log_2(n+1) \leq \log_2(n) + 1 &\iff \\ \log_2(n+1) \leq \log_2(2n) &\iff \\ n + 1 \leq 2n &\iff \\ 1 \leq n, & \end{aligned}$$

wobei wir verwendet haben, dass $\log_2$ (streng) monoton wachsend ist. Die Ungleichung ist für alle natürlichen Zahlen $n > 0$ korrekt. Wir berechnen nun

$$\begin{aligned} &\log_2(n+1) \leq \log_2(n) + 1 \Rightarrow \\ &\lceil \log_2(n+1) \rceil \leq \lceil \log_2(n) + 1 \rceil = \lceil \log_2(n) \rceil + 1. \end{aligned}$$

Definition A.2:
Es sei $\Sigma = \{a_1, a_2, \ldots, a_n\}$, $n > 0$, ein Alphabet mit der Ordnung $a_1 < a_2 < \ldots < a_n$. Es seien x, y beliebige Wörter über Σ. Wir definieren die kanonische Ordnung $<$ auf allen Wörter über Σ wie folgt:

1. Falls x kürzer ist als y, dann gilt $x < y$.
2. Sind anderenfalls x und y gleich lang, dann gilt $x < y$ genau dann, wenn x alphabetisch vor y liegt.

A.2 Code

```
from time import time

import matplotlib.pyplot as plt
import numpy as np
from numba import jit


# JIT-compiled Mandelbrot iteration function
@jit(nopython=True)
def mandelbrot(c, max_iter):
    z = 0
    for n in range(max_iter):
        if abs(z) > 2:
            return n
        z = z * z + c
    return max_iter


# Generate the Mandelbrot set
```

```
@jit(nopython=True, parallel=True)
def generate_mandelbrot(width, height, re_start, re_end, im_start, im_end,
    max_iter):
    image = np.zeros((height, width), dtype=np.float32)
    for x in range(width):
        for y in range(height):
            re = re_start + (x / width) * (re_end - re_start)
            im = im_start + (y / height) * (im_end - im_start)
            c = complex(re, im)
            image[y, x] = mandelbrot(c, max_iter)
    return image


# Configuration
width, height = 15360, 8640  # 16K resolution
re_start, re_end = -2.0, 1.0
im_start, im_end = -1.0, 1.0
max_iter = 500

# Timing and execution
print("Generating Mandelbrot set...")
start = time()
image = generate_mandelbrot(width, height, re_start, re_end, im_start, im_end,
    max_iter)
end = time()
print(f"Done in {end - start:.2f} seconds.")

# Save the image
output_file = "mandelbrot_16k_hsv.png"
plt.imsave(output_file, image, cmap="hsv")
print(f"Saved image to {output_file}")
```

Programm A.1: `mandelbrot.py`

Literatur

[1] Jürgen Schmidhuber. *LOW-COMPLEXITY ART*. 1994. URL: https://sferics.idsia.ch/pub/juergen/locoart.pdf.