Datenkomprimierung und Huffman-Kodierung

1 Motivation

Für die digitale Verarbeitung werden Daten aller Art, wie z.B. Texte, Bilder, Audiosignale und Videos digital repräsentiert bzw. *kodiert*. Je nach Art der gewählten Kodierung werden für eine gegebene Menge an Informationen mehr oder weniger Bits benötigt, um diese zu repräsentieren. Kompakte (also wenig redundante) Kodierungen haben den Vorteil, dass sie das Datenvolumen minimieren, was eine effizientere Speicherung und Übertragung von Daten ermöglicht. Hingegen können redundantere Kodierungen eine schnellere Verarbeitung und die Fehlererkennung sowie –korrektur ermöglichen.

2 Ziele

Das Ziel dieser Unterrichtssequenz ist es, die Grundidee der Datenkomprimierung einzuführen und mit der Huffman-Kodierung eine optimale Kodierung für Texte vorzustellen. Die Schülerinnen und Schüler

- verstehen die Motivation für die Datenkomprimierung,
- lernen, dass verschiedene Kodierungen der gleichen Information zu unterschiedlich langen Darstellungen führen und somit unterschiedlich grosse Datenvolumina benötigen,
- können Texte mit vorgegebenen Kodierungen kodieren und dekodieren,
- verstehen den Zusammenhang zwischen der relativen Häufigkeit von Zeichen in einem Text und der Länge ihrer Kodierungen,
- verstehen den Vorteil der Präfixfreiheit von Kodierungen,
- kennen die Baumdarstellung von Kodierungen und können damit Texte kodieren und dekodieren,
- können den Huffman-Algorithmus für eine gegebene Häufigkeitsverteilung anwenden.

3 Vorwissen

Die Unterrichtssequenz richtet sich an Schülerinnen und Schüler im Ergänzungsfach. Als Vorwissen wird vorausgesetzt:

- Binärdarstellung von Zahlen
- Eine Standardkodierung von Zeichen, z.B. ASCII

- Präfix und Suffix von Symbolfolgen (Wörtern)
- In der Mathematik:
  - Grundlagen der Kombinatorik
  - Absolute und relative Häufigkeit, Häufigkeitsverteilung
  - Gewichteter Durchschnitt (Erwartungswert in der Wahrscheinlichkeitslehre)
- Grundlagen der Programmierung in Python, insbesondere:
  - Objektorientierung
  - Rekursive Datenstrukturen

4 Unterrichtssequenz

4.1 Einführung

In einem Unterrichtsgespräch klären wir die Motivation für eine kompakte Repräsentation von Daten wie z.B. von Bildern oder Videos. Dabei knüpfen wir an die Erfahrungen der Schülerinnen und Schüler (SuS) an, die sowohl mit dem begrenzten Speicherbedarf ihrer Mobiltelefone vertraut sind als auch mit der teilweise langsamen Datenübertragung im Mobilfunknetz. Ziel ist es, dass die SuS erkennen, dass es erstrebenswert ist, Informationen mit einem möglichst geringen Datenvolumen zu repräsentieren.

Im nächsten Schritt erarbeiten wir mit der Klasse Möglichkeiten, dies zu erreichen. Die SuS werden bereits wissen, dass man die Datenmenge reduzieren kann, indem man z.B. die Grösse von Bildern und Videos oder ihre Auflösung reduziert. Beides führt jedoch zu einem Informations- (d.h. Qualitäts-)verlust. Anhand eines einfachen Beispiels erarbeiten wir, dass es auch Möglichkeiten gibt, die Datenmenge zu verkleinern, ohne die Menge an Information zu reduzieren. Dazu betrachten wir die Kodierung von Bildern.

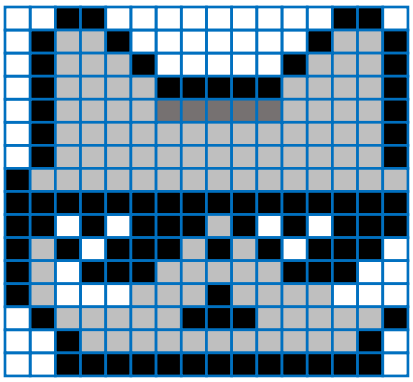
Beispiel: Nehmen wir an, wir *kodieren* ein Graustufen-Bild der Grösse 64x64 Pixel, indem wir für jeden Bildpunkt eine Graustufe zwischen 0 (schwarz) und 255 (weiss) abspeichern. Mit einem Byte (8bit) speichern wir $2^8=256$ unterschiedliche Inhalte und somit die Graustufe eines Pixels. Wenn wir eine feste Ordnung der Pixel betrachten (z.B. zeilenweise), dann benötigt das Bild also 4KB (64x64byte).

Definition: Kodierung
Eine Kodierung ist eine Informationsdarstellung als Folge von Symbolen eines Alphabets (z.B. Buchstaben oder Nullen und Einsen). Normalerweise ist es immer möglich, aus der kodierten Information das dargestellte Objekt eindeutig und vollständig zu erzeugen; dieser Prozess heisst Dekodierung.

Viele Bilder enthalten grössere einfarbige Flächen, z.B. Gebäude oder einen Himmel. D.h., sie wiederholen die gleiche Graustufe (oder auch Farbe) viele Male hintereinander. Diese Eigenschaft kann man ausnutzen, um Bilder effizienter zu kodieren.

In der sogenannten Lauflängenkodierung speichern wir immer abwechselnd zwei Zahlen: Die erste gibt die Graustufe an, die zweite, wie oft diese Graustufe wiederholt werden soll. Im Extremfall – für ein vollständig einfarbiges Bild – braucht diese Kodierung für ein Bild der Grösse 64x64 nur 32byte (wir wiederholen 16 mal 256 Punkte einer Graustufe). Somit haben wir eine kompaktere Darstellung der gleichen Information gefunden: Wir haben das Bild *komprimiert*.

Definition: Komprimierung
Eine Komprimierung (oder Datenkompression) strebt an, eine kürzere Kodierung für eine gegebene Menge von Objekten (Buchstaben, Texte, Bilder, Videos) zu finden.



Aufgabe 1:

Gegeben ist dieses 16x16 Bild. Wie viel Speicher wird benötigt, wenn wir pro Pixel ein Byte pro Graustufe verwenden? Wie viel Speicher wird mit der Lauflängenkodierung benötigt? Gehen Sie davon aus, dass wir die Pixel zeilenweise von links nach rechts durchlaufen.

4.2 Textkodierung

Eine Lauflängenkodierung wäre für Texte nicht zielführend, da Buchstaben in Wörtern typischerweise nicht häufig hintereinander vorkommen – die Lauflänge ist also meistens 1, sodass sich der Speicherbedarf sogar erhöht.

Aufgabe 2:

Betrachten Sie die folgende Liste mit Vornamen und die ihnen üblicherweise zugewiesenen Spitznamen. Was fällt Ihnen auf?

Vorname	Spitzname	Vorname	Spitzname
Peter	Pit	Elisabeth	Elli
Michael	Michi, Mike	Susanne	Susi
Thomas	Tom	Gabriele	Gabi
Andreas	Andi	Christine	Christa
Christian	Chris	Barbara	Babs
Johann	Hans	Martina	Tina
Alexander	Alex	Katharina	Kathi
Friedrich	Fritz	Cornelia	Conny

Auch in Texten kommen manche Buchstaben deutlich häufiger vor als andere (z.B. ‚e', ‚n', ‚i', ‚s' im Vergleich zu ‚j', ‚y', ‚x', ‚q') – wir kennen das von Spielen wie Scrabble oder Hangman. Wir können versuchen, eine kompakte Kodierung für Buchstaben zu finden, indem wir „Spitznamen" für häufige Zeichen nutzen, d.h., die häufigen Buchstaben mit kürzeren und die seltenen mit längeren Bitfolgen darstellen.

Aufgabe 3:

Nehmen wir an, wir hätten ein Alphabet mit nur vier Buchstaben (wie das z.B. bei der DNS der Fall ist, in der es vier verschiedene Basen gibt).

Wir betrachten zwei verschiedene Kodierungen für diese vier Buchstaben:

Standardkodierung: A: 00 B: 01 C: 10 D: 11
Kompakte Kodierung: A: 0 B: 10 C: 110 D: 111

- Kodieren Sie das „Wort" ACAABDA in diesen beiden Kodierungen. Wie lang sind die resultierenden Bitfolgen jeweils?

Die Standardkodierung ist analog zu Kodierungen wie ASCII oder Unicode: Sie benutzt gleich viele Bits pro Zeichen, unabhängig davon, wie häufig ein Zeichen üblicherweise vorkommt. Die kompakte Kodierung benutzt einen sehr kurzen Code für ‚A', dafür einen längeren für ‚C' und ‚D'. Das führt zu einer kürzeren Bitfolge, falls ‚A' deutlich häufiger vorkommt als ‚C' und ‚D', was in unserem Beispiel der Fall ist. Es gibt auch Wörter, für die die kompakte Kodierung deutlich schlechter abschneidet als die Standardkodierung, z.B. ACDC (8 bzw. 10bit).

4.3 Präfixfreiheit und Baumdarstellung

Aufgabe 4:
Betrachten wir wieder das Alphabet {A, B, C, D} und die kompakte Kodierung von oben.

- Dekodieren Sie die Bitfolge 010100.

Nun versuchen wir, eine noch kompaktere Kodierung zu finden.

Neue Kodierung: A: 0 B: 10 C: 01 D: 1

Aufgabe 5:

- Dekodieren Sie die Bitfolge 010100 mit dieser Kodierung.

An der vorherigen Aufgabe sehen wir, dass wir für das Dekodieren wissen müssen, wo die Kodierung eines Buchstabens aufhört und die des nächsten beginnt. In der Standardkodierung ist das trivial: Jeder Buchstabe ist mit genau zwei Bits kodiert (analog bei ASCII). Bei der „neuen Kodierung" wissen wir das nicht. Nachdem wir eine ‚0' gelesen haben, wissen wir nicht, ob wir diese als ‚A' dekodieren sollen oder ggf. gemeinsam mit einer nachfolgenden ‚1' als ‚C'. Wegen dieser Unsicherheit konnten wir in der vorherigen Aufgabe keine eindeutige Dekodierung finden.

Bei der kompakten Kodierung ist das anders.

Kompakte Kodierung: A: 0 B: 10 C: 110 D: 111

Wenn wir eine ‚0' lesen, handelt es sich definitiv um ein ‚A', weil die Kodierung keines anderen Buchstabens mit einer ‚0' beginnt. Ist das erste Bit eine ‚1', müssen wir definitiv weiterlesen. Nun wiederholt sich die Abfolge: Kommt nach der ‚1' eine ‚0', handelt es sich definitiv um ein ‚B', weil die Kodierung keines anderen Buchstabens mit „10" beginnt. Ist das zweite Bit jedoch auch eine ‚1', dann müssen wir definitiv das dritte Bit lesen, um zu wissen, um welchen Buchstaben es sich handelt.

Die hier beschriebene Eigenschaft heisst *Präfixfreiheit*: Keine zwei Zeichen werden so kodiert, dass eine Kodierung ein Präfix der anderen ist. Dadurch wissen wir immer, ob wir nach einem gelesenen Bit das Zeichen dekodieren können oder weiterlesen müssen. Die kompakte Kodierung ist präfixfrei, die „neue Kodierung" jedoch nicht, z.B. weil die Kodierung von ‚A' (‚0') ein Präfix der Kodierung von ‚C' („01") ist.

Aufgabe 6:

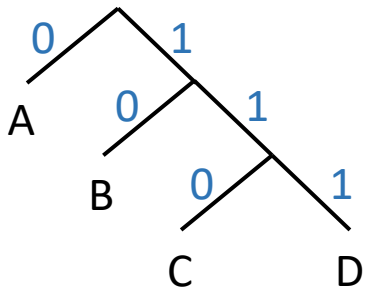
Ist folgende Kodierung präfixfrei?

A: 0 B: 10 C: 110 D: 101

Aufgabe 7:

Finden Sie eine präfixfreie Kodierung für ein Alphabet mit den Buchstaben {A, B, C, D, E, F}, in der nicht alle Buchstaben die gleiche Kodierungslänge haben.

Für präfixfreie Kodierungen können wir den Prozess des Dekodierens als Entscheidungsbaum darstellen (hier am Beispiel unserer kompakten Kodierung):



An jedem Knoten lesen wir entweder eine ‚0' oder eine ‚1'. Führt die entsprechende Kante zu einem Buchstaben, können wir diesen dekodieren (der Pfad zum Buchstaben gibt seine Kodierung an). Andernfalls lesen wir weiter. Dass die Kodierung präfixfrei ist, sehen wir daran, dass wir nur an den Blättern des Baumes Buchstaben vorfinden.

Umgekehrt dient der Baum auch zum Kodieren: Um einen Buchstaben zu kodieren laufen wir von der Wurzel zu diesem Buchstaben; seine Kodierung sind die Nullen und Einsen entlang der durchlaufenen Kanten.

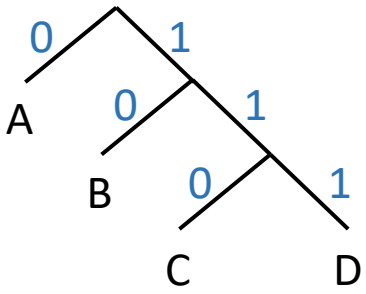
Aufgabe 8:

Stellen Sie folgende präfixfreie Kodierung als Baum dar.

A: 00 B: 01 C: 100 D: 101 E: 110 F: 111

4.4 Häufigkeitsverteilungen und Güte der Kodierung

Wenn wir wissen, wie häufig jeder Buchstabe relativ zu den anderen vorkommt und wie jeder Buchstabe kodiert wird, können wir ausrechnen, wie viele Bits wir im Schnitt pro Buchstabe brauchen. Nehmen wir unsere kompakte Kodierung und folgende Häufigkeit der vier Buchstaben.



A	B	C	D
60%	20%	10%	10%

Im Durchschnitt benötigen wir in diesem Fall je Zeichen 60% x 1bit + 20% x 2bit + 10% x 3bit + 10% x 3bit = 0.6 x 1bit + 0.2 x 2bit + 0.1 x 3bit + 0.1 x 3bit =1.6bit (da 1% = 0.01). Daran sehen wir, dass die kompakte Kodierung für einen Text mit vielen ‚A' und wenigen ‚C' und ‚D' *besser* ist als die Standardkodierung, die genau zwei Bits pro Zeichen braucht.

Definition: Güte einer Kodierung
Eine Kodierung ist besser als eine andere, wenn sie für eine gegebene Häufigkeitsverteilung im Durchschnitt je Zeichen weniger Symbole (Bits) braucht.

Aufgabe 9:

Betrachten wir die folgenden beiden Kodierungen für das Alphabet {A, B, C, D, E, F} und folgende Häufigkeitsverteilung.

Kodierung 1: A: 0 B: 10 C: 110 D: 1110 E: 11110F: 11111

Kodierung 2: A: 00 B: 01 C: 100 D: 101 E: 110 F: 111

A	B	C	D	E	F
30%	25%	15%	10%	15%	5%

Wie viele Bits pro Zeichen brauchen diese beiden Kodierungen für Texte mit den angegebenen relativen Häufigkeiten der einzelnen Zeichen?

Aufgabe 10:

Stellen Sie Kodierung 1 als Baum dar. Vergleichen Sie den Baum mit dem von Kodierung 2 (siehe vorherige Übungsaufgabe). Was fällt ihnen auf?

4.5 Huffman-Kodierung

Wir führen einen Wettbewerb durch, bei dem jeder versucht, eine möglichst gute Kodierung zu finden.

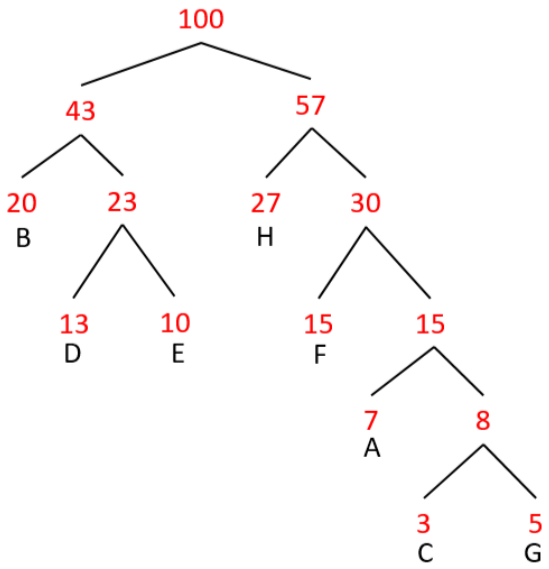
Aufgabe 11:

Finden Sie eine möglichst gute Kodierung für die folgende Häufigkeitsverteilung. Wie viele Bits brauchen Sie im Durchschnitt pro Buchstabe?

A	B	C	D	E	F	G	H
7%	20%	3%	13%	10%	15%	5%	27%

In unserem Wettbewerb haben wir festgestellt, dass es nicht einfach ist, von Hand gute Kodierungen zu finden. Daher drängt sich die Frage auf, ob es ein systematisches Verfahren (einen Algorithmus) gibt, um die optimale Kodierung für eine gegebene Häufigkeitsverteilung zu finden. Die Antwort ist „ja": Die Huffman-Kodierung ist ein solcher Algorithmus.

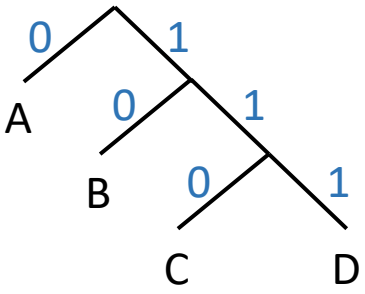
Wir haben bereits gesehen, dass es von Vorteil ist, wenn unser Baum balanciert ist (siehe Aufgabe 10). Dabei müssen wir natürlich die Häufigkeiten der Buchstaben berücksichtigen. Hierfür ist es hilfreich, an jedem Baumknoten zu notieren, wie gross die *Gesamthäufigkeit* aller Buchstaben in diesem Teilbaum ist. Der Baum rechts zeigt das. Die Gesamthäufigkeiten sind in roter Farbe notiert; die Nullen und Einsen haben wir weggelassen, weil sie keine wesentliche Rolle spielen (wir können z.B. davon ausgehen, dass die linke Kante immer mit ‚0' und die rechte immer mit ‚1' markiert ist).



Wir sehen drei wesentliche Eigenschaften in diesem Baum:

- Die Häufigkeiten der Blätter können wir der Häufigkeitsverteilung der Buchstaben entnehmen.
- Die Gesamthäufigkeit jedes inneren Baumknotens ist die Summe der Gesamthäufigkeiten seiner Kinder.
- Die Gesamthäufigkeit der Wurzel ist immer 100%, da der Baum alle Buchstaben enthält.

Gegeben ist folgende Kodierung mit der zugehörigen Häufigkeitsverteilung.



A	B	C	D
60%	20%	10%	10%

Aufgabe 12:

Ergänzen Sie den Baum um die jeweiligen Gesamthäufigkeiten.

Die Huffman-Kodierung erzeugt einen Baum, in dem die Gesamthäufigkeiten optimal balanciert sind und daher die durchschnittliche Anzahl Bits pro Buchstabe minimal ist. Dazu geht sie wie folgt vor:

Eingabe: Ein Alphabet von Zeichen sowie deren relative Häufigkeitsverteilung.

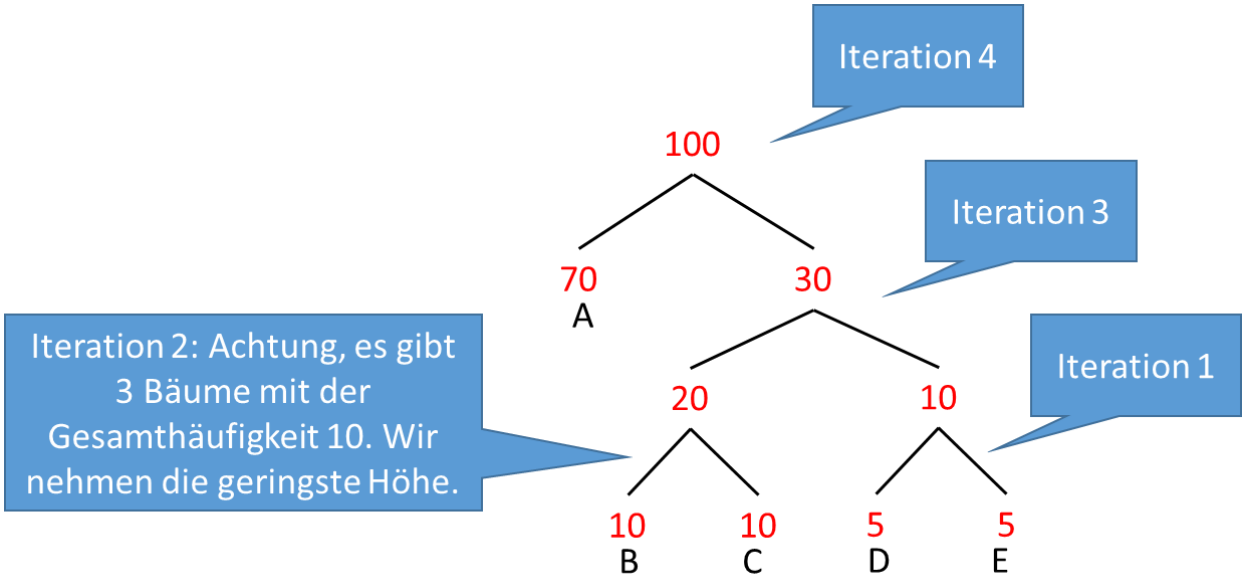
Ausgabe: Eine optimale Kodierung als Baum.

Der Algorithmus nutzt als Datenstruktur eine Menge von Bäumen. Jeder Knoten speichert seine Gesamthäufigkeit sowie die Menge der Buchstaben in diesem Teilbaum (diese Information wird für eine effiziente Kodierung benötigt). Jedes Blatt speichert lediglich seinen Buchstaben.

1. Start:
   Zu Beginn befindet sich in der Menge ein Baum für jeden Buchstaben. Dieser besteht nur aus einem Blatt. Seine Gesamthäufigkeit ist die gegebene relative Häufigkeit.
2. Wiederholen Sie, solange die Menge mindestens zwei Bäume enthält:
   a. Wählen Sie zwei Bäume B1 und B2 aus, deren Wurzeln möglichst kleine Gesamthäufigkeiten haben. Wenn die Auswahl nicht eindeutig ist, dann wählen Sie die Bäume mit der geringsten Höhe.
   b. Verbinden Sie B1 und B2, indem Sie einen neuen Baum B erzeugen, dessen Kinder B1 und B2 sind. Seine Gesamthäufigkeit ist die Summe der Gesamthäufigkeiten von B1 und B2 und seine Buchstabenmenge die Vereinigung der Buchstabenmengen von B1 und B2.
   c. Ersetzen Sie B1 und B2 in der Menge durch B.

Wir illustrieren den Algorithmus Schritt für Schritt an folgendem Beispiel; die Buchstabenmengen in den inneren Baumknoten lassen wir einfachheitshalber weg:

A	B	C	D	E
70%	10%	10%	5%	5%



Versuchen wir, intuitiv zu verstehen, warum die Huffman-Kodierung optimal ist. Sie erzeugt einen Kodierungsbaum, der gleichmässig ausbalanciert ist. D.h., für jeden inneren Knoten des Baumes versucht der Algorithmus dafür zu sorgen, dass die Summe der relativen Häufigkeiten der Zeichen im linken und rechten Teilbaum möglichst gleich ist. Durch diese Balancierung wird der Kodierungsbaum eher breit als tief, sodass die Pfade von der Wurzel zum Blatt (also die Länge der Codes) möglichst kurz ist.

Während des Ablaufs des Algorithmus wächst der Kodierungsbaum von unten nach oben, weil wir immer existierende Teilbäume zu einem Baum zusammenfügen, indem wir sie unter einem neuen Knoten einfügen. Daher landen die Zeichen, die zuerst verarbeitet werden, besonders weit unten im Baum, haben also die längsten Kodierungen. Aus diesem Grund wählt der Algorithmus in Schritt 2a immer als Nächstes diejenigen Teilbäume mit der geringsten relativen Gesamthäufigkeit.

Zurück zu unserem Wettbewerb!

Aufgabe 13:

Benutzen Sie die Huffman-Kodierung, um eine optimale Kodierung für diese Häufigkeitsverteilung zu finden.

A	B	C	D	E	F	G	H
7%	20%	3%	13%	10%	15%	5%	27%

5 Zusammenfassung und Ausblick

Wir haben gesehen, dass es möglich ist, durch geeignete Kodierungen Texte so zu repräsentieren, dass sie möglichst wenig Speicherplatz beanspruchen. Allerdings hängt die Kodierung vom jeweiligen Text (insbesondere dessen Häufigkeits-verteilung) ab. Daher kann ein Text nur dekodiert werden, wenn man seine spezifische Kodierung kennt. Andere Kodierungen wie z.B. ASCII sind hingegen universell, d.h., man verwendet die gleiche Kodierung für alle Texte.

6 Programmieraufgabe für Motivierte

Aufgabe 14:

Erstellen Sie ein Python-Programm, das als Eingabe eine Häufigkeitsverteilung nimmt und die dazugehörige Huffman-Kodierung berechnet und ausgibt. Die folgenden Teilaufgaben führen Sie Schritt für Schritt zu einer Lösung.

Teilaufgabe 14a:

Erstellen Sie eine Python-Klasse Tree, welche den Kodierungsbaum repräsentiert. Jeder Knoten des Baums speichert die Daten, die wir bei der Besprechung des Huffman-Algorithmus gesehen haben. Allerdings genügt es für diese Aufgabe, wenn Sie die Buchstaben in den Blättern speichern; die Buchstabenmengen der inneren Knoten lassen wir zur Vereinfachung weg.

Erstellen Sie einen Konstruktor __init__ für Ihre Klasse, der ein Blatt erzeugt. Der Konstruktor nimmt als Parameter den Buchstaben für dieses Blatt und seine relative Häufigkeit als Zahl zwischen 0 und 1.

Testen Sie Ihre Implementierung mit diesem main-Programm. Es erzeugt keine Ausgabe.

```
def main():
        baum = Tree('A', 0.25)
```

Teilaufgabe 14b:

Wir wollen später das Ergebnis der Huffman-Kodierung als Text ausgeben. Dazu fügen wir unserer Klasse Tree zwei Methoden hinzu, deren Implementierungen Sie schreiben sollen:

```
def get_string(self, praefix)
```

und

```
def __str__(self)
```

Die Methode get_string durchläuft den Baum rekursiv bis zu jedem Blatt und sammelt dabei die Kodierung für jedes Zeichen im Parameter praefix auf. D.h., wenn die Methode zum linken Teilbaum geht, fügt sie dem String praefix eine '0' hinzu, beim rechten Teilbaum eine '1'. Wenn sie bei einem Blatt ankommt, beinhaltet praefix die Kodierung des dortigen Zeichens. Die Methode get_string gibt einen String zurück, der die gesamte Kodierung enthält (siehe Beispiel unten).

Die Methode __str__ benutzt einfach get_string mit leerem praefix.

Testen Sie Ihre Implementierung mit diesem main-Programm:

```
def main():
    b1 = Tree('A', 0.25)
    b2 = Tree('B', 0.35)
    b = Tree(None, 0.6)
    b.links = b1
    b.rechts = b2
    b.hoehe = 1
    print(b)  # ruft implizit __str__ auf
```

Es erzeugt diese Ausgabe:

```
A: 0, B: 1
```

Teilaufgabe 14c:

Der Huffman-Algorithmus verknüpft immer wieder zwei bestehende Bäume, indem er sie als Teilbäume in einen neuen Baum einfügt. Um das zu ermöglichen, fügen Sie Ihrer Tree-Klasse folgende Methode hinzu:

```
def merge(self, rechts)
```

Sie liefert als Ergebnis einen neuen Knoten, der die Parameter self bzw. rechts als Teilbäume hat. Achten Sie darauf, die Felder des neuen Knotens (hoehe, gesamthaeufigkeit) richtig zu setzen.

Testen Sie Ihre Implementierung mit diesem main-Programm.

```
def main():
    b1 = Tree('A', 0.25)
    b2 = Tree('B', 0.35)
    b3 = Tree('C', 0.4)
    b4 = b1.merge(b2)
    b = b4.merge(b3)
    print(b)
    print(b.hoehe)
    print(b.gesamt)
```

Es erzeugt diese Ausgabe:

```
A: 00, B: 01, C: 1
2
1.0
```

Teilaufgabe 14d:

Der Huffman-Algorithmus wählt jeweils die beiden Teilbäume mit der geringsten Gesamthäufigkeit aus (und die weniger hohen bei gleicher Gesamthäufigkeit). Daher bietet es sich an, dass wir im Algorithmus alle Bäume immer sortiert speichern. Zu diesem Zweck implementieren wir den „kleiner"-Operator für Bäume. Fügen Sie dazu Ihrer Klasse Tree die folgende Methode hinzu („lt" steht für „less than"):

```
def __lt__(self, other)
```

Sie soll True zurückgeben, falls der Baum self eine echt kleinere Gesamthäufigkeit als der Baum other hat oder falls beide gleich sind, aber self echt weniger hoch ist als other.

Testen Sie Ihre Implementierung mit diesem main-Programm.

```
def main():
    b1 = Tree('A', 0.25)
    b2 = Tree('B', 0.35)
    print(b1 < b2)
```

Es erzeugt diese Ausgabe:

```
True
```

Teilaufgabe 14e:

Nun sind wir in der Lage, den Huffman-Algorithmus zu implementieren. Schreiben Sie dazu eine Methode ausserhalb der Klasse Tree:

```
def huffman(haeufigkeit)
```

Der Parameter haeufigkeit ist ein Python Dictionary, dessen Schlüssel die Buchstaben und dessen Werte die jeweiligen relativen Häufigkeiten sind. Die Menge aller Schlüssel ist damit unser Alphabet. Dadurch können wir z.B. eine Häufigkeitsverteilung für das Alphabet {A, B, C} wie folgt definieren:

```
h = {'A': .25, 'B': .35, 'C': .4}
```

Die relative Häufigkeit können wir dann mittels der get-Methode abfragen. So ergibt z.B. der Aufruf h.get('B') den Wert .35.

Um die Teilbäume während der Ausführung des Huffman-Algorithmus sortiert zu speichern, benutzen wir Pythons Datenstruktur heapq. Dafür müssen Sie diese wie folgt importieren:

```
from  heapq import *
```

Folgende drei Operationen sind wichtig für uns:

```
baeume = []  # erzeugt eine leere Liste von Bäumen

heappush(baeume, b)       # fügt Baum b sortiert in diese Liste ein

r = heappop(baeume)       # gibt den kleinsten Baum in der Liste zurück
                            (basierend auf unserer Definition von __lt__)
```

Damit sind Sie nun in der Lage, den Huffman-Algorithmus zu implementieren.

Testen Sie Ihre Implementierung mit diesem main-Programm.

```
def main():
    h = {'A': .25, 'B': .35, 'C': .4}
    b = huffman(h)
    print(b)
```

Es erzeugt diese Ausgabe:

```
C: 0, A: 10, B: 11
```

Teilaufgabe 14f:

Verwenden Sie nun Ihre Implementierung, um die Huffman-Kodierung für das Beispiel aus Aufgabe 13 zu erstellen. Die gegebene Verteilung kann einfach zu Beginn der Main-Methode festgelegt werden (so wie in Teilaufgabe 14e); Sie müssen also keine Eingabe programmieren.

Das Ergebnis kennen Sie ja bereits:

B: 00, E: 010, D: 011, H: 10, F: 110, A: 1110, C: 11110, G: 11111

7 Lösungen

Aufgabe 1: Es braucht 256byte mit der Standardkodierung, aber nur 178byte mit der Lauflängenkodierung, also 30% weiniger, ohne Qualität einzubüssen.

Aufgabe 2: Wir beobachten, dass Spitznamen in der Regel verkürzte Formen der Vornamen sind. Es ist praktisch, für Personen und Dinge, über die wir häufig sprechen, kurze Bezeichnungen zu haben.

Aufgabe 3: (Die Leerzeichen dienen nur der besseren Lesbarkeit):

Standardkodierung:	00 10 00 00 01 11 00 (14bit)
Kompakte Kodierung:	0 110 0 0 10 111 0 (12bit)

Aufgabe 4: ABBA

Aufgabe 5: Es gibt keine eindeutige Lösung. Mögliche Lösungen sind ADADAA, CCAA, ABBA und viele andere. Bei der Sammlung der Ergebnisse in der Klasse werden wir feststellen, dass verschiedene SuS verschiedene Lösungen haben.

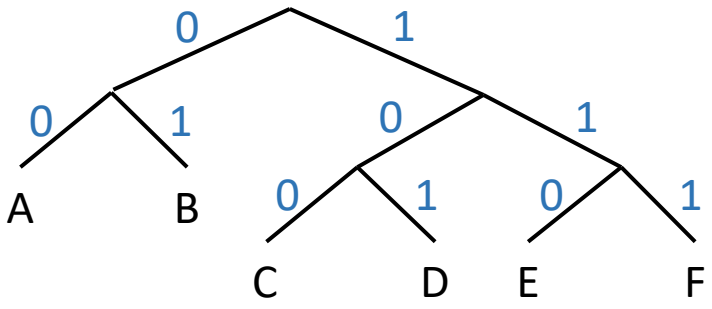
Aufgabe 6: Nein, die Kodierung von ‚B' („10") ist ein Präfix der Kodierung von ‚D' („101"). Stellen Sie sich vor, Sie erhalten den Code „10110". Diesen können Sie nicht eindeutig dekodieren; er könnte für „DB" oder für „BC" stehen. Präfixfreie Kodierungen ermöglichen eine eindeutige Dekodierung.

Aufgabe 7: Mögliche Lösungen sind:

A:	000	B: 001	C: 010	D: 011	E: 100	F: 101
A:	0	B: 10	C: 110	D: 1110	E: 11110	F: 11111
A:	00	B: 01	C: 100	D: 101	E: 110	F: 111

(und viele andere)

Aufgabe 8:



Aufgabe 9:

Kodierung 1:

30% x 1bit + 25% x 2bit + 15% x 3bit + 10% x 4bit + 15% x 5bit + 5% x 5bit =

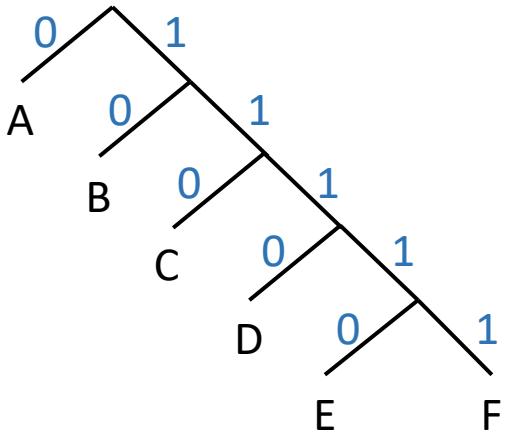
0.3 x 1bit + 0.25 x 2bit + 0.15 x 3bit + 0.1 x 4bit + 0.15 x 5bit + 0.05 x 5bit = 2.65bit

Kodierung 2:

30% x 2bit + 25% x 2bit + 15% x 3bit + 10% x 3bit + 15% x 3bit + 5% x 3bit =

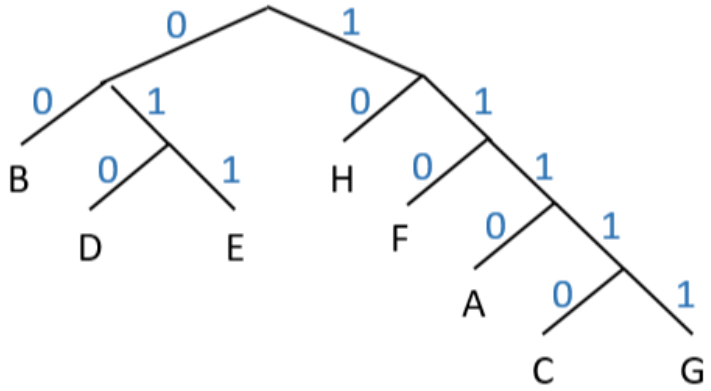
0.3 x 2bit + 0.25 x 2bit + 0.15 x 3bit + 0.1 x 3bit + 0.15 x 3bit + 0.05 x 3bit = 2.45bit

Aufgabe 10:



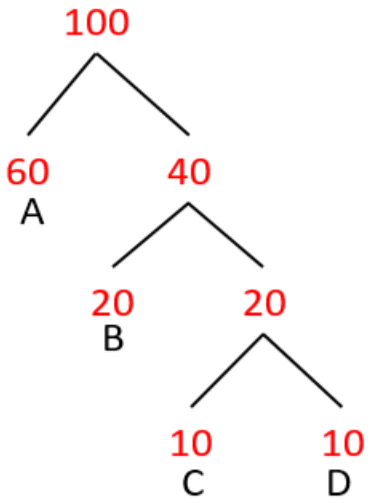
Der Baum für Kodierung 2 ist weniger hoch (Höhe 3 statt 5), d.h., die längsten Kodierungen sind deutlich kürzer als bei Kodierung 1. Da die zugehörigen Buchstaben nicht extrem selten sind (15% und 5%), führt das zu einer effizienteren Kodierung.

Aufgabe 11:



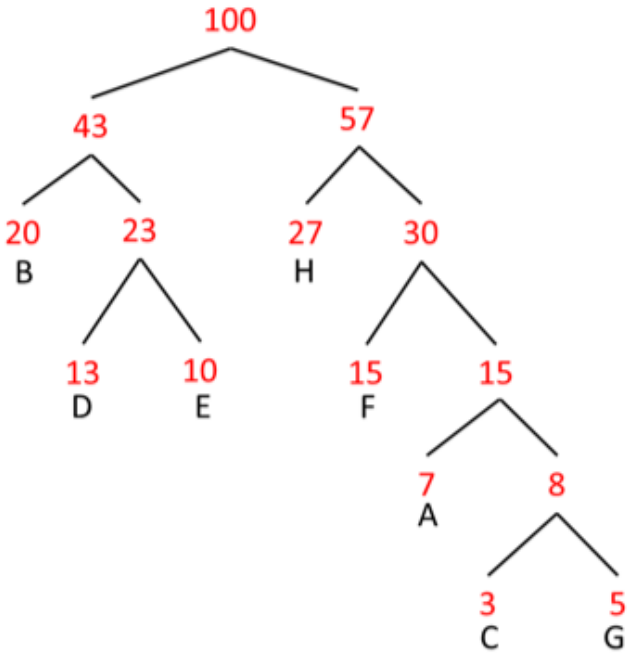
Mit dieser Kodierung brauchen wir im Schnitt 3,22bit pro Buchstabe. Für diese Häufigkeitsverteilung ist die Kodierung eindeutig (natürlich können wir Nullen und Einsen vertauschen, aber es ist klar, welcher Buchstabe wie viele Bits braucht).

Aufgabe 12:



Aufgabe 13:

Wir haben den optimalen Baum bereits gesehen.



Aufgabe 14:

Teilaufgabe 14a:

```
class Tree:
    # Der Konstruktor erzeugt ein Blatt.
    def __init__(self, zeichen, haeufigkeit):
        self.zeichen = zeichen
        self.gesamt = haeufigkeit
        self.hoehe = 0

if __name__ == "__main__":
    main()
```

Teilaufgabe 14b:

```
    # Zur Ausgabe der resultierenden Kodierung. Das Präfix repräsentiert
    # den Pfad oberhalb des Knotens.
    def get_string(self, praefix):
        if self.hoehe == 0:
            return self.zeichen + ': ' + praefix
        else:
            l = self.links.get_string(praefix + '0')
            r = self.rechts.get_string(praefix + '1')
            return l + ', ' + r


    # Zur Ausgabe der resultierenden Kodierung:
    def __str__(self):
        return self.get_string('')
```

Teilaufgabe 14c:

```
    # merge liefert einen neuen Knoten, der die Parameter self und rechts als Kinder hat.
    def merge(self, rechts):
        res = Tree(None, self.gesamt + rechts.gesamt)
        res.hoehe = 1 + (self.hoehe if self.hoehe > rechts.hoehe else rechts.hoehe)
        res.links = self
        res.rechts = rechts
        return res
```

Teilaufgabe 14d:

```
    # Wir definieren den <-Operator, damit wir Bäume in einer
    # Priority Queue speichern können.
    # Die Ordnung ist von der Huffman-Kodierung vorgegeben:
    # (Gesamthäufigkeit, Höhe)
    def __lt__(self, other):
        if self.gesamt < other.gesamt:
            return True
```

```
        if self.gesamt == other.gesamt:
            return self.hoehe < other.hoehe
        return False
```

Teilaufgabe 14e:

```
from  heapq import *

# Diese Methode berechnet die Huffman-Kodierung für eine gegebene Häufigkeitsverteilung.
def huffman(haeufigkeit):
    # Schritt 1: Erzeuge eine Menge von Blättern, eines für jedes Zeichen.
    # Wir verwenden als Datenstruktur eine Priority Queue, die so
    # sortiert ist, dass wir immer die ersten beiden Bäume verknüpfen können
    # (siehe <-Operator in Tree).
    baeume = []
    for zeichen in haeufigkeit.keys():
        baum = Tree(zeichen, haeufigkeit.get(zeichen))
        heappush(baeume, baum)

    # Schritt 2: Finde die Huffman-Kodierung.
    while len(baeume) >= 2:
        l = heappop(baeume)
        r = heappop(baeume)
        n = l.merge(r)
        heappush(baeume, n)

    # Ergebnis: der finale Baum
    return heappop(baeume)
```

Teilaufgabe 14f:

```
def main():
    # Eingabe: Die relativen Häufigkeiten sind zwischen 0 und 1;
    # ihre Summe muss 1 ergeben.
    haeufigkeit = {'A': .07, 'B': .2, 'C': .03, 'D': .13, 'E': .1, 'F': .15, 'G': .05, 'H': .27}
    # Ein weiterer Testfall:
    # haeufigkeit = {'A': .7, 'B': .1, 'C': .1, 'D': .05, 'E': .05}

    # Kodierung berechnen
    res = huffman(haeufigkeit)

    # Ausgabe: Die resultierende Kodierung
    print(res)
```