Stefan Kurtz
Emanuel Ehmki

Universität Hamburg
Zentrum für Bioinformatik

Programmierung für Naturwissenschaften 1
Wintersemester 2019/2020
Übungen zur Vorlesung: Ausgabe am 18.12.2019

... und immer wieder am Mittwoch verfügbar:
`https://feedback.informatik.uni-hamburg.de/PfN1/wise2019-2020`

Aufgabe 9.1 (6 Punkte) Eine Permutation einer Menge S ist eine Liste der Elemente aus S, in der jedes Element genau einmal vorkommt. Jede Permutation ist also eine Liste der Länge n, wenn n die Anzahl der Elemente in S ist. Verschiedene Permutationen von S unterscheiden sich durch die Reihenfolge der Elemente aus S. Die Menge der Permutationen von S wird durch $Perms(S)$ bezeichnet.

Beispiel: Für $S = \{0, 1, 2\}$ ist $Perms(S) = \{[2, 1, 0], [1, 2, 0], [2, 0, 1], [0, 2, 1], [1, 0, 2], [0, 1, 2]\}$.

Für Permutationen gibt es vielfältige Anwendungen in der Mathematik und Informatik.

Ein einfacher rekursiver Algorithmus zur Berechnung von $Perms(S)$ funktioniert nach den folgenden Regeln:

- Falls $S = \emptyset$, dann ist $Perms(S) = \{[\,]\}$, d.h. die leere Liste ist die einzige Permutation der leeren Menge S.
- Falls $S = \{a\}$ (d.h. S besteht aus genau einem Element a), dann ist $Perms(S) = \{[a]\}$.
- Falls S mindestens zwei Elemente enthält, dann berechnet man für alle $a \in S$ die Menge $R(S, a) = Perms(S \setminus \{a\})$.[1] $Perms(S)$ besteht in diesem Fall aus den Listen $p.append(a)$ für alle $a \in S$ und $p \in R(S, a)$.

Beispiel: Set $S = \{0, 1, 2\}$. Nach dem obigen Verfahren berechnet man zunächst die Mengen $R(S, 0)$, $R(S, 1)$ und $R(S, 2)$. Es ist $R(S, 0) = Perms(S \setminus \{0\}) = Perms(\{1, 2\})$. Daher berechnet man zunächst die Mengen $R(\{1, 2\}, 1)$ und $R(\{1, 2\}, 2)$. Es gilt:

$$R(\{1, 2\}, 1) = Perms(\{2\}) = \{[2]\}$$
$$R(\{1, 2\}, 2) = Perms(\{1\}) = \{[1]\}$$

Damit ergibt sich $R(S, 0) = \{[2, 1], [1, 2]\}$. Analog erhält man $R(S, 1) = \{[2, 0], [0, 2]\}$ und $R(S, 2) = \{[1, 0], [0, 1]\}$. Aus $R(S, 0)$ berechnet man durch Anhängen von 0 die beiden Permutationen $[2, 1, 0], [1, 2, 0]$. Aus $R(S, 1)$ berechnet man durch Anhängen von 1 die beiden Permutationen $[2, 0, 1], [0, 2, 1]$. Aus $R(S, 2)$ berechnet man durch Anhängen von 2 die zwei Permutationen $[1, 0, 2], [0, 1, 2]$. Damit wurden alle $3! = 6$ Permutationen von S berechnet.

Benennen Sie die Datei `allperms_template.py` in `allperms.py` um. Implementieren Sie darin eine Python-Funktion `all_permutations(elems)`, die die Liste aller Permutationen der Elemente aus `elems` nach dem obigen Algorithmus berechnet und als `return`-Wert zurückliefert.

[1] Der Operator $\setminus$ steht für die Mengendifferenz, d.h. für zwei Mengen A und B ist $A \setminus B = \{a \mid a \in A, a \notin B\}$.

Die Menge der Elemente aus S bzw. ihre Teilmengen können Sie jeweils als Liste darstellen, d.h. `elems` ist eine Liste. In der Implementierung müssen Sie Elemente aus Listen löschen. Durch `l.pop(idx)` können Sie das Element an Index `idx` in einer Liste `l` löschen.

Die obige Beschreibung legt eine rekursive Berechnung nah. Daher implementieren Sie die obige Funktion auf der Basis einer rekursiven Funktion
`all_perms_rec(all_perms,elems)`, die in der Liste `all_perms` die Permutationen der Liste `elems` berechnet. `all_perms_rec` hat keinen Rückgabewert.

Beachten Sie, dass eine Wertzuweisung `p = q` für zwei Listen `p` und `q` eine Referenz `p` auf die Liste `q` erzeugt. Da `all_perms_rec` mit verschiedenen Listen `elems` aufgerufen wird, müssen Listen kopiert werden. `q.copy()` liefert eine Kopie der Liste `q`.

Schreiben Sie außerdem eine Funktion `all_permutations_verify(all_perms,elems)`, die verifiziert, dass die Liste `all_perms` die Liste aller Permutationen von `elems` ist. Dabei sind die Elemente in `elems` aufsteigend sortiert. Sei n die Anzahl der Elemente in `elems`. In der Funktion `all_permutations_verify` müssen Sie mit Hilfe von `assert` die folgenden Bedingungen verifizieren:

- Die Länge von `all_perms` ist $n!$.
- Es gibt keine Permutation in `all_perms`, die mehr als einmal vorkommt.
- Wenn man die einzelnen Elemente aus `all_perms` (also die Listen) jeweils sortiert, ergibt sich immer die Liste `elems`.

Durch die Verwendung von `assert` bricht das Programm ab, wenn eine der genannten Bedingungen nicht zutrifft.

Im Hauptprogramm werden die beiden genannten Funktionen für Listen der Länge $i, 0 \leq i \leq n$ aufgerufen, wobei n das einzige Argument des Hauptprogramms ist. Durch `make test` verifizieren die Korrektheit Ihrer Implementierung.

Aufgabe 9.2 (5 Punkte) In dieser Aufgabe geht es darum, in Python3 eine Klasse `Morse` zur Codierung und Decodierung eines Textes durch Morse-Zeichen zu implementieren. Für jedes alphanumerische Zeichen sowie die Zeichen `.` und `,` ist der Morse-Code eine Folge zweier Signallängen (kurz und lang, dit und dah im englischen). Diese Signallängen werden durch die Zeichen `.` und − beschrieben. Einen String, der nur aus den Zeichen `.` und − besteht, nennen wir *Morse-String*.

In der Datei `morseClass_template.py` finden Sie eine Basis-Implementierung der Klasse `Morse` mit einem Dictionary `morse_code`, das die Codierung der genannten Zeichen in einen Morse-String definiert. `morse_code` enthält zusätzlich eine Codierung für das Leerzeichen. Benennen Sie die Datei `morseClass_template.py` um in `morseClass.py`.

Ihre Aufgabe besteht aus den folgenden Teilaufgaben.

1. Implementieren Sie in der Klasse `Morse` eine Methode `encode(self,text)`, die einen String `text` als Argument erhält und mit einer `return`-Anweisung den entsprechenden Morse-String zurückliefert. Bei der Codierung sollen Kleinbuchstaben wie die entsprechenden Großbuchstaben behandelt werden. Während bei der traditionellen Anwendung von Morse-Codes zwischen der Codierung von zwei aufeinanderfolgenden Zeichen eine kurze Pause erfolgt, die man in einem String typischerweise durch ein Leerzeichen codiert, soll das in Ihrer Implementierung nicht erfolgen. Beispiel: Für den String `SOS` liefert die Funktion den Morse-String `...---...` 1 Punkt

2. Wenn das Programm `morseClass.py` mit der Option `--text` aufgerufen wird, wird der Morse-String angezeigt. Im Makefile sind einige Tests implementiert, die testen, ob Ihre Funktion `encode` korrekt funktioniert.
3. Wenn das Programm `morseClass.py` nicht mit der Option `--text` oder `--decode` aufgerufen wird, dann wird ein Shell-Skript generiert, das entsprechend der Zeichen des Morse-Strings ein Programm zum Abspielen der Dateien `dit.wav` und `dah.waw` aufruft. Wenn man dieses Shell-Skript über eine Pipe mit dem Befehl `sh -s` verbindet (siehe Makefile), kann man den Morse-String hören. Dafür muss unter macOS das Programm `afplay` und unter Linux das Programm `aplay` verfügbar sein.[2] Wenn das bei Ihnen der Fall ist, können Sie sich die Morsezeichen anhören (falls der Lautsprecher an ist). Testen Sie die Funktionalität durch den Aufruf von `make test_sound`.
4. Auch wenn man den Morse-Code kennt, kann man den durch die Funktion `encode` gelieferten Morse-String nicht immer eindeutig decodieren. Geben Sie eine Erklärung hierfür. Konstruieren Sie ein Beispiel, an dem Sie das Problem verdeutlichen. Dazu müssen Sie sich in `morseClass.py` das Dictionary `morse_code` ansehen. 1 Punkt
5. Um eine eindeutige Decodierung eines Morse-Strings zu ermöglichen, muss man eine andere Codierung verwenden, wie z.B. die Codierung entsprechend dem Dictionary `morse_code2_0` . Wenn man die Option `--mc2_0` wählt, erfolgt die Codierung entsprechend diesem Dictionary. Beispiel: Der Code von `SOS` entsprechend `morse_code2_0` ist `..-.-.....-.`
6. Implementieren Sie in der Klasse `MorseClass` eine Methode `decode`, die einen Morse-String (entsprechend der Codierung mit `morse_code2_0`) als Argument erhält und den hierdurch codierten Text mit einer `return`-Anweisung als Ergebnis liefert. Beispiel: Für den obigen Morse-String `..-.-.....-.` liefert diese Funktion den decodierten String `SOS`. 2 Punkte
7. In den Materialien finden Sie zwei Dateien `unknown_short.code2_0` und `unknown.code2_0` mit Morsestrings (Version 2.0) für zwei unbekannte Texte. Die Originaltexte sind nicht Teil der Testdaten. Trotzdem kann durch den Aufruf von `make test_decode` getestet werden, ob Ihre Implementierung der Methode `decode` korrekt funktioniert. Beschreiben Sie, warum das möglich ist. Dafür müssen Sie sich das Makefile ansehen und den Optionsparser ansehen und ausprobieren, was die einzelnen Kommandos beim Aufruf von `make test_decode` bewirken. Schauen Sie sich außerdem den Text an, der durch `./morseClass.py -d unknown.code2_0` ausgegeben wird. Der Text erläutert eine wichtige Eigenschaft von Codierungen. 1 Punkt

Bitte die Lösungen zu diesen Aufgaben bis zum 06.01.2020 um 18:00 Uhr an pfn1@zbh.uni-hamburg.de schicken. Die Besprechung der Lösungen erfolgt am 08.01.2020.

[2] Wenn Sie ein anderes Programm, wie z.B. `cvlc` verwenden möchten, müssen Sie in der Funktion `play_morse` die Zeile mit `SOUNDPLAYER=` entsprechend ändern.