Informatik I: Einführung in die Programmierung

Prof. Dr. Bernhard Nebel
Dr. Stefan Wölfl, Thorsten Engesser
Wintersemester 2015/2016

Universität Freiburg
Institut für Informatik

Übungsblatt 4

Abgabe: Freitag, 20. November 2015, 20:00 Uhr

Wichtige Hinweise: Zur Bearbeitung der Übungsaufgaben legen Sie bitte ein neues Unterverzeichnis `sheet04` im Wurzelverzeichnis Ihrer Arbeitskopie des SVN-Repositories an. Ihre Lösungen werden dann entsprechend dem ersten Übungsblatt in Dateien in diesem Unterverzeichnis erwartet.

Beachten Sie bitte bei allen Aufgaben die *Hinweise zur Bearbeitung der Übungsaufgaben* unter der folgenden URL:

`http://gki.informatik.uni-freiburg.de/teaching/ws1516/info1/wiki/hinweise.html`

Überprüfen Sie, dass Sie alle Lösungen ins Repository hochgeladen haben (z.B. mit dem Befehl `svn status`). Überprüfen Sie auch die Webseite Ihres SVN-Unterverzeichnisses:

`https://daphne.informatik.uni-freiburg.de/ws1516/InformatikI/svn/$RZLOGIN`

Bewertet wird bei allen Aufgaben die letzte Version, die zur Deadline des Übungsblattes auf dem SVN-Server eingereicht ist.

Aufgabe 4.1 (Permutationen; Datei: `permutation.py`; Punkte: 3+3+3+3)

Unter einer *n-Permutation* (für $n \in \mathbb{N}$) verstehen wir im Folgenden eine bijektive Abbildung $\sigma : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$. Intuitiv lassen sich Permutationen als Vertauschungsvorschrift interpretieren (siehe `https://de.wikipedia.org/wiki/Permutation`). Die *Identitätspermutation* ist hierbei eine besondere Permutation, die jedes Element aus der Menge $\{1, 2, \ldots, n\}$ auf sich selbst abbildet.
In den folgenden Teilaufgaben werden wir typische Operationen für Permutationen implementieren. Dazu werden wir n-Permutationen in der üblichen Tupelschreibweise als Python-Tupel der Länge n repräsentieren: ein Python-Tupel $(k_0, k_1, \ldots, k_{n-1})$, in dem alle k_i aus $\{1, \ldots, n\}$ sowie paarweise verschieden sind, repräsentiert die n-Permutation σ, die der Zahl 1 die Zahl k_0, der Zahl 2 die Zahl k_1, etc., zuordnet.

(a) Durch (einmalige) *Anwendung einer n-Permutation σ auf eine Sequenz (der Länge n)* $\overline{x} = x_1, x_2, \ldots, x_n$ erhält man die Sequenz $\overline{x}^{\sigma} = x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)}$.

Implementieren Sie in der Datei `permutation.py` eine Funktion `apply(s, lst)`, die eine n-Permutation `s` auf eine Liste `lst` der Länge n einmalig anwendet. Die Funktion soll die eingegebene Liste `lst` verändern und `None` zurückgegeben.

Betrachten Sie dazu folgendes Beispiel:

```
>>> sigma = (4, 3, 2, 1)
>>> spam = ['s', 'p', 'a', 'm']
>>> apply(sigma, spam)
>>> spam
['m', 'a', 'p', 's']
```

```
>>> sigma = (1, 3, 4, 2)
>>> spam = ['s', 'p', 'a', 'm']
>>> apply(sigma, spam)
>>> spam
['s', 'a', 'm', 'p']
```

(b) Die *Komposition* zweier n-Permutationen σ und τ ist definiert als die n-Permutation, die zuerst τ und dann σ ausführt; formal: $(\sigma \circ \tau)(x) := \sigma(\tau(x))$.

Implementieren Sie in der Datei `permutation.py` eine Funktion `compose(s, t)`, die die Komposition zweier n-Permutationen `s` und `t` zurückgibt. Betrachten Sie dazu folgende Beispiele:

```
>>> sigma = (1, 3, 4, 2)
>>> tau = (3, 1, 2, 4)
>>> compose(sigma, tau)
(4, 1, 3, 2)
>>> compose(tau, tau)
(2, 3, 1, 4)
```

(c) Die *Ordnung* einer n-Permutation σ ist definiert als die kleinste natürliche Zahl $k > 0$, so dass die k-fache Komposition von σ die Identitätspermutation $(1, 2, \ldots, n)$ ergibt. Die k-fache Komposition von σ ist hierbei rekursiv wie folgt definiert: $\sigma^1 := \sigma$ und $\sigma^{k+1} := (\sigma \circ \sigma^k)$.

Implementieren Sie eine Funktion `order(s)`, die bei Eingabe einer n-Permutation `s` die Ordnung von `s` zurückgibt. Betrachten Sie dazu folgende Beispiele:

```
>>> sigma = (2, 1, 4, 3)
>>> order(sigma)
2
>>> sigma = (2, 3, 4, 1)
>>> order(sigma)
4
```

(d) Die *inverse Permutation* zu einer n-Permutation σ ist definiert als die n-Permutation τ, für die sowohl $(\sigma \circ \tau)$ als auch $(\tau \circ \sigma)$ die Identitätspermutation $(1, 2, \ldots, n)$ ergeben.

Implementieren Sie eine Funktion `inverse(s)`, die das Inverse einer Permutation `s` zurückgibt (*Hinweis:* In der Zweizeilenform einer Permutation lässt sich die inverse Permutation darstellen, indem man die zwei Zeilen vertauscht und die Spalten umsortiert; siehe den referenzierten Wikipedia-Artikel). Betrachten Sie dazu folgende Beispiele:

```
>>> sigma = (2, 3, 1, 4)
>>> sigmai = inverse(sigma)
>>> sigmai
(3, 1, 2, 4)
>>> compose(sigma, sigmai)
(1, 2, 3, 4)
>>> compose(sigmai, sigma)
(1, 2, 3, 4)
```

Aufgabe 4.2 (Nim-Spiel; Datei: `nim.py`; Punkte: 6)

Wir betrachten eine einfache Variante des Spiels *Nim*. Bei Spielanfang liegen $k \in \mathbb{N}$ Streichhölzer auf dem Tisch. Es gibt zwei Spieler, die abwechselnd 1, 2 oder 3 Streichhölzer wegnehmen dürfen. Wer das letzte Streichholz vom Tisch nimmt, gewinnt.

Laden Sie sich zunächst das Template `nim.py` von der Übungswebseite herunter. Dieses enthält eine Simulation des Spiels, die Sie über einen Aufruf von `python3 nim.py` in der Konsole starten können. Die Simulation erlaubt es, auf der Konsole gegen den Computer zu spielen. Allerdings trifft der Computergegner seine Entscheidungen momentan noch per Zufall. Ihre Aufgabe ist es, die Implementierung des Computergegners stattdessen mit einer optimalen Strategie auszustatten, d.h. einer Strategie, die in jedem Spielzustand, in dem es für den ziehenden Spieler möglich ist, den Gewinn des Spiels zu garantieren, eine entsprechende Aktion auswählt.

Implementieren Sie dazu die Strategiefunktion `nim_strategy(s)`, die für einen Spielzustand (repräsentiert durch die Anzahl s der auf dem Tisch verbliebenen Streichhölzer) entweder die Anzahl der optimalerweise zu ziehenden Streichhölzer zurückgibt, oder `False`, sofern das Spiel gegen einen optimal spielenden Spieler nicht mehr gewonnen werden kann. Greifen Sie dabei **nicht** auf die einfache Lösungsvorschrift zurück, die sie auch online finden können, sondern implementieren Sie die Funktion *rekursiv*: Überlegen Sie sich dazu zunächst, in welchen Zuständen das Spiel verloren ist oder mit einer Aktion gewonnen werden kann. Für andere Zustände soll Ihre Funktion für die verschiedenen Aktionsmöglichkeiten prüfen, ob nach der jeweiligen Aktion ein Zustand erreicht ist, in dem ein optimaler Gegner den Gewinn garantieren kann.

Hinweis: Wenn Sie die Funktion richtig implementiert haben, kann ein Nutzer Ihres Programms (zumindest für den vorgegebenen Parameter $k = 24$) nicht mehr gewinnen.

Aufgabe 4.3 (Erfahrungen; Datei: `erfahrungen.txt`; Punkte: 2)

Legen Sie im Unterverzeichnis `sheet04` eine Textdatei `erfahrungen.txt` an. Notieren Sie in dieser Datei kurz Ihre Erfahrungen beim Bearbeiten der Übungsaufgaben (Probleme, benötigter Zeitaufwand nach Teilaufgabe, Interessantes, etc.).