Stefan Kurtz,
Dirk Willrodt

Universität Hamburg
Zentrum für Bioinformatik

Programmierung in der Bioinformatik
Wintersemester 2013
Übungen zur Vorlesung: Ausgabe am 16.12.2013

Mit diesem Übungsblatt möchten wir Ihnen schöne Ferien wünschen und einen guten Rutsch in das Jahr 2014.

Aufgabe 10.1 Gegeben sei eine Datei, in der jede Zeile nach dem folgenden Schema aufgebaut ist:

A=k B=x C=y D=z

Dabei sind k, x, y, z ganze Zahlen. Zwischen den vier einzelnen Blöcken können beliebig viele Leerzeichen stehen. Wir nennen k den Schlüssel der Zeile und x, y, z die Werte der Zeile. Sie sollen nun ein Ruby-Programm schreiben, das solche Zeilen zerlegt, die Summen der Werte einer Zeile berechnet und diese unter dem Schlüssel der Zeile abspeichert. Insgesamt sollen für alle Zeilen der Datei die Summen für einen Schlüssel zu einer Gesamtsumme zusammengefaßt und am Ende ausgeben werden. Ihr Programm soll robustes Verhalten zeigen, d.h. bei Fehleingaben oder falsch formatierten Dateien eine sinnvolle Fehlermeldung ausgeben und beenden. Falls der Aufruf korrekt ist, zum Beispiel mit der Datei `sumzeile.data`, die unter Stine zur Verfügung steht, dann wird das Ergebnis folgendermaßen ausgegeben:

```
sum[4] = 6284
sum[20] = 5971
sum[50] = 5862
sum[80] = 5897
```

Was passiert bei Ihrem Programm, wenn man sehr große Werte für k in der Datei hat?

Aufgabe 10.2 Bei der Translation werden die auf der mRNA codierten genetischen Informationen zur Synthese der entsprechenden Proteine genutzt. Eine codierende Sequenz beginnt in der Regel mit dem Startcodon (AUG) und endet mit einem Stopcodon (UAA, UAG oder UGA). Bei Prokaryoten treten oft polycistronische mRNA-Stränge auf, bei denen auf einer mRNA mehrere codierende Bereiche zu finden sind, welche unterschiedlichen Proteinen entsprechen.

Schreiben Sie ein Programm, welches eine mRNA-Sequenz aus einer Fasta-Datei einliest und die codierten und die untranslatierten Regionen (UTR) der mRNA bestimmt. Identifizieren Sie dazu die Start- und Stopcodons mit Hilfe von möglichst kurzen regulären Ausdrücken und achten Sie darauf, dass die erkannten Stopcodons im Bezug auf die jeweiligen Startcodons im richtigen Leseraster liegen. Startcodons können hingegen an jeder beliebigen Position auftreten. Gehen Sie außerdem davon aus, dass in der mRNA-Sequenz keine Überschneidung von codierenden Bereichen auftritt.

Sie können für die Eingabedatei annehmen, dass diese nur einen Fasta-Eintrag enthält. Ihr Programm kann aber auch alle Einträge in einer Fasta-Datei bearbeiten.

Als Ergebnis soll das Programm die eingelesene mRNA Sequenz so ausgeben, dass untranslatierte Bereiche durch Klein-, codierende Sequenzen hingegen durch Großbuchstaben gekennzeichnet werden.

Geben Sie auch die Fasta-Header mit aus, besonders wenn Sie Dateien mit mehreren Einträgen zulassen.

```
Beispiel:
  Eingabe: ACGUGCUAUGCGGAGGGCAGUCUAACAUCGGAGAAAAAAAAAAAAA
  Ausgabe: acgugcuAUGCGGAGGGCAGUCUAAcaucggagaaaaaaaaaaaaa
```

Das Stopcodon zählen wir hier zur codierenden Sequenz.

Verwenden Sie Ihr Programm mit der in Stine bereitgestellten Eingabedatei `mRNA.fas` und vergleichen Sie Ihre Ausgabe mit der Ausgabedatei `mRNA_output.dat`.

Aufgabe 10.3 In dieser Aufgabe geht es darum, in Ruby eine erste eigene Klasse zu implementieren, nämlich eine Klasse für Stacks (Stapel).

In einem Stack kann man Daten mit dem FILO-Prinzip ablegen. FILO steht hierbei für „First In Last Out“, d.h., dass die Daten, die man zuerst auf den Stack gelegt hat, als letztes wieder entfernt werden.

Auf Ihrem Stack sollen folgende Operationen möglich sein.

`push(x)` Legt das Element `x` auf den Stack.

`pop()` Nimmt das oberste Element vom Stack.

`top()` Liefert das oberste Element des Stacks zurück, ohne es vom Stack zu holen.

`empty?` Liefert genau dann `true` zurück, wenn der Stack leer ist.

Sie finden unter Stine eine Datei `stack-skeleton.rb`, die als Grundgerüst für die Implementierung des Stack dient.

Ergänzen Sie die Datei so, dass der Test in der Datei `stackTest.rb` ohne Fehler durchläuft. Dafür müssen Sie Ihre Stack-Implementation in der Datei `stack.rb` abspeichern und `stackTest.rb` ausführen.

Beantworten Sie nun folgenden Fragen:

- Was bedeutet das `@`-Zeichen vor der Variablen `data`? Kann man außerhalb der `class` Umgebung darauf zugreifen?
- Was passiert, wenn man auf einem leeren Stack `pop()` ausführt? Wie kann der Programmierer darauf reagieren, bzw. was passiert in der Funktion `test`, die in `stackTest.rb` definiert ist?
- Entfernen Sie die `rescue`-Anweisung. Was passiert nun, wenn Sie das Skript `stackTest.rb` ausführen?

Die Lösungen zu diesen Aufgaben werden am 13.01.2014 besprochen.