Bonus. Übung

Abgabe bis 07.07.2017, 10:00 Uhr

Einzelaufgabe Bonus.1: U2I2 — 20 EP

Obwohl ein Großteil der I2-Homepage schon automagisch mit Daten aus dem UnivIS generiert wird, mussten $\mathcal{JohnDoe}$ und $\mathcal{M}ka$ die Stundenpläne der AuD- bzw. PFP-Übungen bislang noch immer von Hand erstellen. Helfen Sie ihnen, auch diesen Schritt zu automatisieren. Laden Sie U2I2.zip von der AuD-Website und machen Sie sich mit den enthaltenen Dateien vertraut.
Ihre Aufgabe ist es, eine Unterklasse von `U2I2Abstract` namens `U2I2` zu implementieren, die einen AuD/PFP-Stundenplan als HTML-Dokument im übergebenen Objekt `htmlI2` erstellt. Dazu bekommt Ihr Programm ein `U2I2Config`-Objekt, das den zu generierenden Stundenplan beschreibt (z.B. welche Tage und Uhrzeiten darzustellen sind), und ein Feld mit XML-Dokumenten, die die Daten der zugehörigen Veranstaltungen aus dem UnivIS enthalten.
Ein XML-Dokument können Sie wie einen Baum traversieren, um Daten aus den Knoten zu extrahieren (innere Knoten tragen Informationen in benannten Attributen, Blätter hingegen als Klartext); exemplarisch sehen Sie das in mehreren vorgegebenen Methoden der Klasse `U2I2Abstract` - öffnen Sie die `*.xml`-Dateien dazu in Ihrem Browser (IE und Firefox stellen sie menschenlesbar dar). Ein HTML-Dokument ist prinzipiell ähnlich aufgebaut. Zur Erstellung von Unterbäumen stehen Ihnen in `U2I2Abstract` Hilfsmethoden zur Verfügung, die ebenda exemplarisch verwendet werden, um das Grundgerüst zusammenzustellen, das Ihre Tabelle umschließen soll.

a) Implementieren Sie (nach dem Konstruktor) zuerst die Methode `collectKnownRooms`, die alle übergebenen XML-Bäume traversiert und die darin gefundenen Räume als $(key, short)$-Paar in `knownRooms` ablegt – z.B. für die Vorlesung als („*Room.nat.dma.zentr.h11*“, „*H11*“).

b) Ergänzen Sie nun die dazu ähnliche Methode `collectKnownPersons`, die aus allen XML-Bäumen die dort aufgeführten Personendaten extrahiert und in `knownPersons` ablegt.

c) Implementieren Sie schließlich die eigentliche Generierung des Stundenplans in der Methode `generateTimeTableHTML`. Die Methode bekommt als Parameter den HTML-Knoten, der die Tabelle darstellt und soll diese nur noch befüllen. Ihr Stundenplan soll keine Fußnoten erzeugen und die Fußnotenzeile muss leer sein, ansonsten soll der generierte Plan ziemlich exakt genauso aussehen, wie in den Mustern WS2014/15 bzw. WS2016/17 aus dem öffentlichen Testfall. Auch hier helfen Ihnen Ihre Browser (u.a. IE und Firefox), indem Sie einzelne HTML-Elemente und deren Darstellung auf Mausklick bidirektional anspringen.

Modifizieren Sie *niemals* die übergebenen XML-Dokumente! Testen Sie Ihren Code *auch* mittels `U2I2PublicTest` (zur korrekten Darstellung benötigen Sie einen Internetzugang zu den CSS-Dateien der AuD-Website) und geben Sie Ihre `U2I2.java` über EST ab.

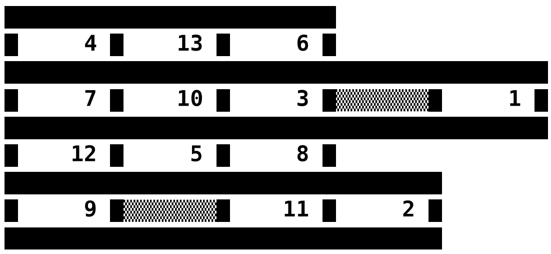
Einzelaufgabe Bonus.2: Das etwas andere *Springerproblem* — 13 EP

Das klassische Springerproblem besteht darin, eine Folge von Rössel-Sprüngen zu finden, so dass das Pferd jedes Feld eines Schachbretts genau einmal besucht. Anstelle eines gewöhnlichen Schachbretts betrachten wir hier „nach rechts ausgefranste Java-Schachbretter“ mit „Lücken“, die *nicht* betreten werden dürfen. Implementieren Sie eine Klasse `JavaSpringerProbleme` mit der Methode

```
int[][] loese(int startSpalte, int startZeile, boolean[][] brett):
```

- Das Java-Schachbrett wird durch `brett` definiert: Bei `false` darf das entsprechende Feld *nicht* betreten werden – andernfalls muss das zugehörige Feld *genau einmal* besucht werden. *Achtung:* Das Schachbrett `brett` darf *niemals nicht* verändert werden!
- Der Springer soll bei `brett[startZeile][startSpalte]` starten: Falls das übergebene Brett gar keine Zeilen hat oder mind. eine Zeile `null` ist oder das geforderte Startfeld ungültig ist, dann muss `loese` wie üblich eine `IllegalArgumentException` werfen.
- Gibt es keine Lösung für das Rätsel, dann muss `loese` einfach `null` zurückgeben.
- Falls es mind. eine Lösung gibt, dann ist eine beliebige davon als 2D-`int`-Feld zurückzugeben. Dieses muss exakt die gleiche Form, wie das `brett`-Feld haben (d.h. die gleiche Anzahl Zeilen bzw. Anzahl Spalten pro entsprechender Zeile). Genau an den Stellen, an denen `brett` eine „Lücke“ (= `false`) aufweist, muss das Ergebnisfeld den Wert 0 haben – alle anderen Felder sind in der Reihenfolge durchnummeriert, in der der Springer sie besucht hat.

Das *kleine Beispiel* aus dem Testfall könnte folgende Lösung ausgeben (benötigt Unicode!):



Einzelaufgabe Bonus.3: Memoization — 12 EP

Sie haben *Dynamische Programmierung* mittels *Memoisation* als mächtiges Mittel kennengelernt, um eine kürzere Laufzeit auf Kosten eines höheren Speicherverbrauchs zu erreichen. Um diesen Vorteil zu nutzen, mussten Sie bislang Ihre naiv-rekursiven Methodenrümpfe umprogrammieren. Mit ein paar Tricks gelingt das auch „nicht-invasiv“, z.B. mit aspektorientierter Programmierung.
Um den Rahmen des Bonusblattes aber nicht zu sprengen, nehmen wir vereinfachend an, dass *alle* Klassen mit einer naiv-rekursiven Methode eine Unterklasse der vorgegebenen Klasse `Rekursion` sind und ihre naiv-rekursiven Methoden ähnlich aussehen, wie in den Beispielen `LongFib` und `BigBinom` im öffentlichen Test. Dabei nimmt die Methode `rekursion` beliebig viele (aber pro Klasse und bei jedem weiteren rekursiven Aufruf stets gleich viele) Parameter vom generischen Typ `I` entgegen und liefert ein Ergebnis vom Typ `O`.
Implementieren Sie eine Klasse `Memoisiert` mit einer Klassenmethode `memoisiere`, die jede `Rekursion` so transformiert, dass sie *dynamische Programmierung* mittels *Memoisation* nutzt, d.h. jeden Aufruf von `rekursion` „abfängt“ und mit einem zwischengespeicherten Wert beantwortet, falls für die gerade übergebenen Aktualparameter schon ein Ergebnis vorliegt.

Einzelaufgabe Bonus.4: Back to the roots — 15 EP

Als Oma zur Schule ging, musste sie sogar Quadratwurzeln noch mit Schiefertafel und Griffel statt mit ihrem iDroid-Smartphone ziehen. Anders als mit dem Smartphone konnte Oma damals schon die Quadratwurzel stellenweise und beliebig genau ausrechnen. Und das ging für $\sqrt{654,321}$ so:

S1: Unterteile die Zahl vom Dezimalpunkt aus in beide Richtungen in Gruppen zu je 2 Ziffern (die erste Gruppe ist ggf. kleiner und die letzte wird mit 0 aufgefüllt): $\sqrt{6|54,32|10|00|\dots}$

S2: Die erste Ziffer der Quadratwurzel ist die größte Zahl a, deren Quadrat gerade noch kleiner oder gleich der ersten Gruppe ist: Da $2^2 \leq 6 < 3^2$ ist, lautet die erste Stelle also $a = 2$.

S3: Das Quadrat dieser ersten Ziffer wird von der ersten Gruppe abgezogen und an das Ergebnis wird die nächste Gruppe $\underline{54}$ angehängt: $b := (6 - a^2) \cdot 100 + \underline{54} = 2\underline{54}$

S4: Nun wird die größte Ziffer c gesucht, für die $(a \cdot 20 + c) \cdot c \leq b$ gerade noch gilt: Wegen $(2 \cdot 20 + 5) \cdot 5 \leq 254 < (2 \cdot 20 + 6) \cdot 6$ lautet die zweite Stelle der Wurzel daher $c = 5$.

S5: Die Zwischenergebnisse b und a werden mit $d := (a \cdot 20 + c) \cdot c$ aus **S4** und der nächsten Gruppe $\underline{32}$ aktualisiert: $b \leftarrow (b - d) \cdot 100 + \underline{32} = 29\underline{32}$ und $a \leftarrow (a \cdot 10 + c) = 25$.

S6: Anschließend wiederholt man die Schritte **S4** und **S5** *ad infinitum*, wobei man nach der letzten Gruppe „beliebig" lange mit $\underline{00}$ auffüllt. Sobald die nächste Gruppe unmittelbar „hinter" dem Dezimalpunkt steht, wird auch in der Lösung der Dezimalpunkt gesetzt.

Der weitere Ablauf ab $a = 25$, $b = 29\underline{32}$ und $\sqrt{6|54,\underline{32}|10} = 25,\ldots$ lautet also:

▶ $(25 \cdot 20 + 5) \cdot 5 \leq 29\underline{32} < (25 \cdot 20 + 6) \cdot 6 \rightsquigarrow c = 5$

▷ $b \leftarrow 407\underline{10}$ sowie $a \leftarrow 255$ $\qquad \sqrt{6|54,32|\underline{10}} = 25,5\ldots$

▶ $(255 \cdot 20 + 7) \cdot 7 \leq 407\underline{10} < (255 \cdot 20 + 8) \cdot 8 \rightsquigarrow c = 7$

▷ $b \leftarrow 4961\underline{00}$ sowie $a \leftarrow 2557$ $\qquad \sqrt{6|54,32|10} = 25,57\ldots$

▶ $(2557 \cdot 20 + 9) \cdot 9 \leq 4961\underline{00} < (2557 \cdot 20 + 10) \cdot 10 \rightsquigarrow c = 9$

▷ $b \leftarrow 35759\underline{00}$ sowie $a \leftarrow 25579$ $\qquad \sqrt{6|54,32|10} = 25,579\ldots$

▶ $(25579 \cdot 20 + 6) \cdot 6 \leq 35759\underline{00} < (25579 \cdot 20 + 7) \cdot 7 \rightsquigarrow c = 6$

▷ $b \leftarrow 506384\underline{00}$ sowie $a \leftarrow 255796$ $\qquad \sqrt{6|54,32|10} = 25,5796\ldots$

...

a) Erstellen Sie eine Klasse `RadicandStream`, welche die Schnittstelle `AbstractStream<T>` geeignet implementiert. Der Konstruktor bekommt ein `BigDecimal`-Objekt und erzeugt den Strom der Zahlenblöcke, wie er in Schritt **S1** bzw. **S6** beschrieben wird.
Die Methode `proceed` muss *genau dann* und damit genau einmal `null` zurückgeben, wenn die nächste Gruppe unmittelbar hinter dem Dezimalpunkt ist. Verwenden Sie dabei die im Englischen übliche Definition („*a period, centered dot, or in some countries a comma at the left of a proper decimal fraction (as .678) or between the parts of a mixed number (as 3.678) expressed by a whole number and a decimal fraction*"), d.h. falls es „keine" Vorkommastelle gibt (z.B. weil die Eingabe positiv aber kleiner 1 ist), dann liefert gleich der erste Aufruf von `proceed` schon mal `null`. Für das obige Beispiel also: $6, 54,$ `null`, $32, 10, 00, 00, \ldots$

b) Erstellen Sie eine Klasse `RootStream`, die ebenfalls die Schnittstelle `AbstractStream<T>` implementiert. Der Konstruktor bekommt auch ein `BigDecimal`-Objekt x und berechnet *nach und nach* den *unendlichen* Strom der einzelnen Stellen von $\sqrt{x}$.
Wie vorhin auch muss die Methode `proceed` *genau dann* und genau einmal `null` zurückgeben, wenn die nächste Ziffer der Quadratwurzel unmittelbar hinter dem Dezimalpunkt ist. Auch hier gilt die gleiche Regel für Dezimalpunkte wie beim `RadicandStream`. Für das obige Beispiel liefert `proceed` also den Strom: $2, 5,$ `null`, $5, 7, 9, 6, 9, 8, 9, 8\ldots$
Achtung: Die Zwischenergebnisse a und b wachsen sehr schnell an. Verwenden Sie daher auch intern *niemals* primitive Datentypen (also *kein* `long` und *kein* `double`), sondern ausschließlich `BigInteger` und `BigDecimal`.

60 EP + 0 GP = 60 Punkte