FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

4. Übung

Abgabe bis 21.11.2016, 10:00 Uhr

Einzelaufgabe 4.1: Rekursionsformen — 12 EP

Legen Sie jeweils fest, welche Rekursionsformen in den folgenden Code-Ausschnitten vorliegen **und begründen** Sie Ihre Antwort.

a)

```
public int binom(int n, int m) {
    if (m == 0 || m == n)
        return 1;
    else
        return binom(n-1, m) + binom(n-1, m-1);
}
```

b)

```
public boolean isBQ1(int n) {
    if (n == 0)
        return true;
    else if (n%2 == 0)
        return isBQ1(n/2);
    else
        return isBQ2(n/2);
}

public boolean isBQ2(int n) {
    if (n == 0)
        return false;
    else if (n%2 == 0)
        return isBQ2(n/2);
    else
        return isBQ1(n/2);
}
```

c)

```
public int div(int x, int y, int r) {
    if (x < y)
        return r;
    else
        return div(x-y, y, r+1);
}
```

d)

```
public int fak(int n) {
    if(n == 0)
        return 1;
    else
        return n * fak(n-1);
}
```

e)

```
public int acker(int m, int n) {
    if (m == 0)
        return m + 1;
    else if (n == 0)
        return acker(m-1, 1);
    else
        return acker(m-1, acker(m, n-1));
}
```

f)

```
public int summeRekursiv(int n) {
    int ergebnis = 0;
    int letzter = n;

    if(n <= 1)
        ergebnis = 1;
    else {
        for(int i = 0; i <= n; i++) {
            ergebnis = letzter + n;
            letzter = ergebnis;
        }
    }

    return ergebnis;
}
```

Geben Sie Ihre Lösung als `Rekursion.pdf` über EST ab.

Einzelaufgabe 4.2: Hanoi — 6 EP

Das Spiel „Türme von Hanoi“ besteht aus drei Stäben „Start“, „Mitte“ und „Ziel“, auf die mehrere gelochte Scheiben gelegt werden, alle verschieden groß. Zu Beginn liegen alle Scheiben auf „Start“, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten oben. Ziel des Spiels ist es, den kompletten Scheiben-Stapel von „Start“ nach „Ziel“ zu versetzen. Bei jedem Zug darf die oberste Scheibe eines beliebigen Stabes auf einen der bei den anderen Stäbe gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe. Folglich sind zu jedem Zeitpunkt des Spieles die Scheiben auf jedem Feld der Größe nach geordnet. Eine interaktive Darstellung des Spieles finden Sie unter `http://www.mathematik.ch/spiele/hanoi_mit_grafik`.

Laden Sie sich die Datei Hanoi.java herunter und implementieren Sie die **rekursive** Methode `solve()`. Als Parameter wird die Anzahl `n` der Scheiben, die zu Beginn des Spieles auf Stab A liegen, übergeben, sowie die Bezeichnungen der drei Stäbe „Start“, „Mitte“ und „Ziel“ als Zeichenkette. Die Rückgabe soll ebenfalls als Zeichenkette erfolgen, die alle durchgeführten Züge beinhaltet. Jeder einzelne Zug soll durch Leerzeichen getrennt im Format `"Start->Ziel"` angegeben werden. Die Lösung für `n=3`, `start="A"`, `mid="B"` und `final="C"` wäre beispielsweise die Zeichenkette `"A->C A->B C->B A->C B->A B->C A->C"`.
Geben Sie Ihre Lösung als `Hanoi.java` über EST ab.

Einzelaufgabe 4.3: Negafibonacci — 7 EP

Die Fibonacci-Reihe wird wie folgt definiert:

$$F_n = F_{n-1} + F_{n-2}$$

Eine mögliche Erweiterung der Fibonacci-Reihe ist die Einbeziehung von negativen Zahlen. Die sogenannte Negafibonacci-Reihe wird wie folgt definiert:

$$F_{-n} = (-1)^{(n+1)} F_n$$

In dieser Aufgabe soll die **rekursive** Berechnung dieser Zahlenfolge implementiert werden. Dabei werden sowohl positive als auch negative Zahlen als Eingabewerte betrachtet. Laden Sie dazu die Datei Negafibonacci.java herunter und vervollständigen Sie die Methode `solve()` der Klasse `Negafibonacci` derart, dass sie für die als Parameter übergebene ganze Zahl `n` das n-te Glied der Folge zurückgibt. Halten Sie sich an folgendes Beispiel:

Tabelle 1: Negafibonacci Reihe Ausschnitt

...	2	-1	1	0	1	1	2	...
...	F_{-3}	F_{-2}	F_{-1}	F_0	F_1	F_2	F_3	...

Geben Sie Ihre Lösung als `Negafibonacci.java` über EST ab.

Gruppenaufgabe 4.4: Merge — 10 GP

Laden Sie sich die Datei Merge.java herunter und ergänzen Sie die darin enthaltene Methode `merge()`. Diese **rekursive** Methode verschmilzt jeweils benachbarte gleiche Zahlen im übergebenen Zahlenarray `ns`, wobei die Verschmelzung mittels Multiplikation erfolgt. Beispielsweise wird aus der Zahlenfolge `ns = 1, 2, 5, 5, 4` das Ergebnis `"1 2 25 4"`. Die Methode startet im Array an der übergebenen Position `i` und arbeitet sich aufsteigend bis zum Ende des Arrays durch. Die Rückgabe soll als Zeichenkette erfolgen, worin die einzelnen Zahlen mit jeweils einem Leerzeichen getrennt sind. Stehen im Array ab dem Index `i` keine Zahlen mehr zur Verfügung, gibt die Methode eine leere Zeichenkette zurück, liegt nur eine Zahl vor, wird nur diese zurückgegeben. Eine Verschmelzung kann somit erst dann erfolgen, wenn mindestens noch zwei Zahlen zur Verfügung stehen. Das Ergebniss bereits verschmolzene Zahlen soll in keine weitere Verschmelzung mit Zahlen eingehen, die dem Ergebniss gleichen, z.B. wird `ns = 2, 2, 4, 6, 8` zu `"4 4 6 8"`. Allerdings werden sämtliche aufeinanderfolgende gleiche Zahlen sehrwohl miteinander verschmolzen, z.B. wird `ns = 5, 5, 5, 6` zu `"125 6"` oder `ns = 2, 2, 2, 8` zu `"8 8"`.
Geben Sie Ihre Lösung als `Merge.java` über EST ab.

Gruppenaufgabe 4.5: Maze — 25 GP

In dieser Aufgabe sollen Sie einen Algorithmus zur Wegfindung in Labyrinthen implementieren. Laden Sie die Dateien `MazeGenerator.java`, `MazeGUI.java` und `Maze.java` herunter. `MazeGenerator` dient zum Erzeugen zufälliger Labyrinthe, die mit Hilfe von `MazeGUI` angezeigt werden können. Hierfür können die Methoden `generate()` zum Erzeugen und `show()` zum Anzeigen von Labyrinthen verwendet werden. Diese beiden Methoden können für spätere Testzwecke hilfreich sein.
Ein Labyrinth wird als 3D-`boolean`-Feld innerhalb der Klasse `Maze` dargestellt. Es ist stets rechteckig, muss aber nicht quadratisch sein. Die erste Dimension des Felds ist der Zeilen- und die zweite der Spaltenindex (siehe auch Abbildung 1). In der dritten Dimension stehen vier `boolean`-Werte, die für jeden der „Räume“ des Labyrinths angeben, in welchen Himmelsrichtungen sich Wände (`false`) und Durchgänge (`true`) befinden. Verwenden Sie zum Zugriff auf die einzelnen Himmelsrichtungen stets die vorgegebenen Konstanten `NORTH, EAST, SOUTH` und `WEST`, **niemals** deren Zahlenwerte. Zum Iterieren über alle Richtungen steht Ihnen auch das Feld `DIRECTIONS` zur Verfügung. Das Labyrinth hat *genau zwei zufällig* gewählte Ein-/Ausgänge.

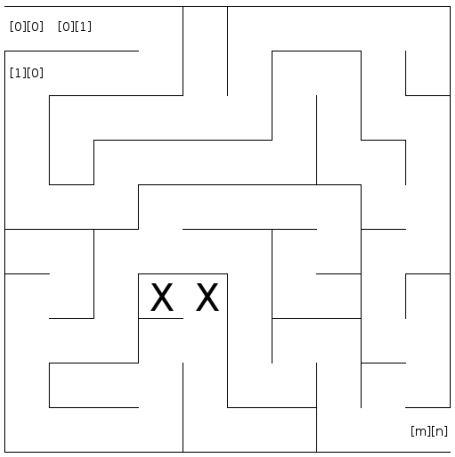

Abbildung 1: Beispiellabyrinth mit 10×10 Räumen und Sackgasse (X)

Da `MazeGenerator` nur Labyrinthe mit genau einem richtigen Weg generiert, besteht eine Lösung darin, alle „Sackgassen“ zu suchen und zu markieren. Durch diesen Vorgang entstehen zunächst immer neue Sackgassen, die ebenfalls markiert werden müssen: Die X-Markierungen in der Abbildung 1 geben beispielhaft gefundene Sackgassen an. Wenn keine neuen Sackgassen mehr markiert werden können, ist nur noch der korrekte Weg durch das Labyrinth unmarkiert. Markierte Sackgassen werden im 2D-`boolean`-Feld `deadEnds` der Klasse `Maze` verwaltet, wobei die erste Dimension der Zeilen- und die zweite Dimenions der Spaltenindex ist. Wird das entsprechende Feld mit `true` markiert, handelt es sich hierbei um eine Sackgasse. Im Folgenden müssen Methoden implementiert werden, die nur im Zusammenspiel die Lösung des Labyrinths ermöglichen. Aus diesem Grund empfielt es sich, den nachfolgenden Text als Ganzes durchzulesen um einen besseren Gesamtüberblick zu bekommen.

a) Vervollständigen Sie die Methode `isDeadEnd()` der Klasse `Maze`. Sie bekommt als Parameter zwei Koordinaten y und x übergeben. Die Methode soll für den Raum an den Labyrinthkoordinaten (y, x) prüfen, ob es sich um eine Sackgasse handelt und dementsprechend `true` oder `false` zurückgeben. Ein Raum gilt nur dann als Sackgasse, wenn er von mind. drei Seiten von Wänden und/oder anderen, schon als Sackgasse identifizierten und in `deadEnds` markierten Räumen umgeben ist. Bitte beachten Sie, dass der Aufruf dieser Methode keine internen Zustände/Felder der Klasse `Maze` verändert.

b) Ergänzen Sie nun die Methode `seekDeadEnd()` der Klasse `Maze`, die das Labyrinth genau einmal in Leserichtung ($\neg\searrow$) durchläuft und die Koordinaten `{y,x}` (als `int[]`-Feld mit genau diesen beiden Werten) des *ersten* gefundenen Raums zurückgibt, der eine bislang nicht markierte Sackgasse ist. Gibt es keine Sackgassen mehr, gibt sie `null` zurück. Bitte beachten Sie erneut, dass der Aufruf dieser Methode keine internen Zustände/Felder der Klasse `Maze` verändert.

c) Vervollständigen Sie nun die Methode `solveMaze()` der Klasse `Maze`. Diese Methode soll dem Aufrufer ein zweidimensionales `boolean`-Feld zurückgeben, so dass an der Feldposition `[y][x]` genau dann `true` steht, wenn der Raum an Koordinate (y, x) Teil einer Sackgasse ist. Für Räume, die nicht zu einer Sackgasse gehören, steht `false` an der entsprechenden Stelle. Verwenden Sie hierfür den Array `deadEnds` der Klasse `Maze` um entsprechende Sackgassen zu makieren.

d) Implementieren Sie als letztes die Methode `getSteps()` der Klasse `Maze`. Diese Methode gibt die minimale Anzahl der Schritte zurück, die für eine Durchquerung des Labyrinths notwendig sind.

Geben Sie Ihre Lösung als `Maze.java` über EST ab.

25 EP + 35 GP = 60 Punkte