Lehrstuhl Computergrafik
Prof. Dr.-Ing. Hendrik P.A. Lensch
Patrick Wieschollek, Sebastian Herholz, Fabian Groh

Eberhard Karls
Universität
Tübingen

Informatik II

Übungsblatt 4

Ausgabe: Di, 10. Mai 2016 - **Abgabe: Di, 24. Mai 2016 - 12:00 Uhr**

Wir möchten noch einmal ausdrücklich betonen, dass InfoMark nur **.zip** Datei entgegennimmt. Lösungen zu den weiteren Aufgaben sind im **pdf**-Format in ihrer Abgabe beizufügen.

Aufgaben

4.1 Black Jack (1 + 2 + 2 + 2 + 2 + 2 + 2 + 2 = 15 Punkte)

Black Jack ist das meistgespielte Karten-Glückspiel in Casinos. Ziel dieser Aufgabe ist es, eine vereinfachte Variante eines interaktiven Black Jack Spiels zu programmieren. Die für uns vereinfachten Regeln sind:

- Ein Spieler spielt gegen die Bank.
- Zu Beginn des Spiels erhalten sowohl der Spieler als auch die Bank eine Karte vom Stapel, welche durch eine zufällige Zahl zwischen 1 und 11 repräsentiert wird.
- Der Spieler darf solange weitere Karten verlangen, bis er glaubt, mit der Summe seiner Kartenwerte nahe genug an die 21 herangekommen zu sein. überschreitet e¿r mit seinen Blatt den Wert 21, so verliert das Spiel sofort. Enthält sein Blatt exakt den Wert 21, so hat er sofort gewonnen.
- Sobald der Spieler keine Karten mehr verlangt ist die Bank an der Reihe. Die Bank zieht nun solange Karten, bis sie mehr oder gleich viel Punkte erreicht hat wie die Spieler. Besitzt das Blatt der Bank einen Wert von mehr als 17 Punkte, so darf diese nicht mehr ziehen. Überschreitet die Bank durch das Ziehen weiterer Karten die 21, so hat sie sofort verloren. Sobald die Bank mehr Punkte als der Spieler in ihrem Blatt besitzt, gewinnt die Bank. Bei gleich vielen Punkten wird das Spiel als Unentschieden gewertet und der Spieler erhält seinen Einsatz zurück.

Gehen Sie wie folgt vor:

a) Fertigen Sie eine Lösungsskizze an und stellen Sie ein Flussdiagramm (Programmablaufplan) auf.

b) Der Spieler hat einen Kontostand und einen Kartenwert (berechnet aus den bisher gezogenen Karten). Die Bank hat einen Kartenwert (berechnet aus den bisher gezogenen Karten).

c) Der Spieler soll am Anfang dazu aufgefordert werden, einen Betrag zu setzen (denken Sie daran, den gesetzten Betrag vom aktuellen Kontostand abzuziehen). Fangen Sie die Fälle ab, bei denen der Spieler ungültige Beträge setzt.

d) Implementieren Sie die Methode `giveCard`, welche eine zufällige Karte, d.h. einen Wert zwischen 1 und 11, zurückgibt.

e) Implementieren Sie die Methode `oneMoreCard`, welche abfragt ob der Spieler eine weitere Karte ziehen will und diese Entscheidung als `boolean` zurückgibt. Beachten Sie den Fall, wenn der Kartenwert des Spielers ≥ 21 ist.

f) Implementieren Sie eine Methode `evaluateWinner`, welche als Argumente die Kartenwerte von Spieler und Bank bekommt und den Gewinner des Spiels als String zurückgibt (”player” vs ”bank” vs ”both”).

g) Implementieren Sie die Methode `updateMoney`, welche als Argumente den aktuellen Kontostand des Spielers, die Höhe des gesetzten Betrags und den Sieger entgegennimmt. Die Methode soll nun den Wert des aktualisierten Kontostands zurückgeben.
Der Spieler bekommt den gesetzten Betrag doppelt gutgeschrieben, wenn er gewonnen hat. Wenn der Spieler verloren hat, bekommt er nichts von seinem gesetzten Betrag zurück.
Sollte das Spiel unentschieden ausgehen, bekommt der Spieler seinen gesetzten Betrag zurück.

h) Das Spiel soll nun über mehrere Runden gehen und mit der `startGame` Methode gestartet werden, welche als Argument das Startguthaben des Spielers bekommt (z.B. 1000). Hat der Spieler am Ende einer Runde noch Guthaben übrig, so wird er gefragt, ob er eine weitere Runde spielen möchte. Ist dies der Fall, so beginnt die nächste Runde, indem der Spieler zunächst wieder gefragt wird, wieviel er dieses Mal setzen möchte. Andernfalls ist das Spiel zu Ende.

Einige Tipps:

- Um eine Eingabe einzulesen, können Sie wieder den `Scanner` verwenden (siehe übungsblatt 1). Mit diesem können Sie den Spieler auch fragen, ob er eine weitere Karte möchte (z.B. Eingabe 1) oder nicht (Eingabe 0). Dokumentieren Sie solche Eigenschaften am Anfang ihres Programms.
- Zum Erzeugen einer Zufallszahl zwischen 1 und 11 können Sie folgendes Codefragment verwenden: `(int)(Math.random()*11+1)`
- Haben Sie eine Endlosschleife programmiert oder möchten Sie das Programm aus anderen Gründen abbrechen, so können Sie dies mit dem roten "Terminate" - Button (in Eclipse! Zu finden ist der rote Quadratische Button unten rechts) oder mit der Tastenkombination `Strg+C` tun.
- Sie können das Programm während der Ausführung jederzeit durch den Befehl `System.exit(0);` stoppen.

4.2 Konfiguration-Tests (5 + 6 + 4= 15 Punkte)

Wir betrachten ein quadratisches Schachbrett der Größe $n \times n$. Ein Feld zu besetzen, heißt dort einen Spielstein zu platzieren. Zwei benachbarte Felder dürfen nicht gleichzeitig besetzt werden. Benachbarte Felder sind jeweils das anschließende linke, rechte, untere sowie obere Feld. Fertigen Sie ein Programm an, welches alle regelkonformen Schachbrettbelegungen für $n = 0, 1, 2, 3, 4, 5, 6$ findet. Zwei Beispiele finden Sie in Abbildung 1. Fügen Sie die Anzahl gefundener gültiger Konfigurationen der Abgabe bei.

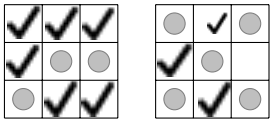

Figure 1: Beispiel zur ungültige Belegung (links) und gültige Belegung (rechts).

Da das direkte Konstruieren von gültigen Lösungen für beliebige n schwierig erscheint, sollten Sie ein Programm erstellen, welches durch schnelles naives Testen aller Zustände gültige Belegungen findet.
Gehen Sie wie folgt vor:

a) Implementieren Sie die Methode `gueltig(boolean[] feld, int n)`, wobei `feld` das Brett mit einer beliebigen Belegung ist und n wieder die Größe des Schachbrettes entspricht. In dieser Methode soll geprüft werden, ob die Belegung im Array `feld` gültig ist und dann entsprechend true/false zurückgegeben werden.

b) Vervollständigen Sie die Methode `moeglichkeiten(int i, int n, int startX, boolean[] feld)`, wobei n wieder die Größe des Schachbrettes definiert. Die Argumente `i` und `startX` sollen die Anzahl der zu legenden Steine, bzw. an welcher Position der aktuelle Stein zu legen ist darstellen. Die Variable `feld` ist ein boolean-Array der Größe $n \times n$, welches das Brett darstellt. Hierbei steht true und false für ein belegtes bzw. freies Feld auf dem Spielbrett.

Diese Methode soll nun (rekursiv) alle möglichen Belegungen im Array `feld` erstellen, unabhängig davon ob sie gültig sind. Jedes mal wenn eine neue Belegung gefunden wurde, wird durch einen

Aufruf von `gueltig` geprüft ob es sich dabei um eine gültige Belegung handelt. Die gültigen Belegungen sollen am Ende zurückgegeben werden.

Tipp: Auch wenn das Array `feld` das Spielbrett darstellen soll, ist es in dieser Methode einfacher sich das Array als ein ein-dimensionales Array vorzustellen.

c) Implementieren Sie die Methode `gueltigeBelegungen(int n)`, wobei n die Größe des Schachbrettes definiert. Diese Methode soll für jede mögliche Anzahl an Steinen die Methode `moeglichkeiten` aufrufen und am Ende die gesamte Anzahl an gültigen Belegungen zurückgeben.

Es macht Sinn - vor dem tatsächlichen Programmieren - sich Gedanken über mögliche Laufzeiten zu machen. Wie viele zulässige Zustände gibt es mindestens für eine beliebige Brettgrößen n? Finden Sie eine nicht-triviale Mindestanzahl von erlaubten Konfigurationen in Abhängigkeit von n. Hierfür kann Ihnen das Betrachten der Abbildung 1 helfen.
Skizzieren Sie grob Ihre Überlegungen, wie man das Problem effizienter lösen könnte.

Einige Hinweise:

- Um die Position von einem Stein auf einem $n \times n$-Brett mit x- und y-Werten zu bestimmen, kann in einem ein-dimensionalen Array folgendermaßen darauf zugegriffen werden: `feld[x*n + y]`.
- Die Variable n kann im Quelltext manuell verändert werden. Der naive Algorithmus benötigt, je nach Programmierung für $n = 6$ etwa 10 min. Dieses Programm für $n = 7$ auszuprobieren ist nicht ratsam.
- Beachten Sie beim Testen, dass sich die Methoden gegenseitig aufrufen. Empfehlenswert ist hier tatsächliche die Methoden in der obigen vorgeschlagenen Reihenfolge zu bearbeiten und zu testen.