Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Jens-Peter Redlich
M. Kabierski, Dr. W. Müller, L. Pfahlsberger, M. Sänger

Berlin, den 28.12.2022

Übungen zur Vorlesung
Grundlagen der Programmierung

Übungsblatt 5

Abgabe: Die Abgabe Ihrer Lösungen erfolgt über den Moodle-Kurs der Vorlesung (https://moodle.hu-berlin.de/enrol/index.php?id=114664). Nutzen Sie die dort im Abschnitt Übung angegebene **Aufgabe 5** und laden Sie insgesamt die **zwei Dateien** `MazeSolver.java` und `Gray.java` **hoch**. Die Abgabe ist bis zum **16.01.2023** um **09:15 Uhr** möglich.

Hinweise:

- Achten Sie darauf, dass Ihre Java-Programmdateien mittels des UTF-8-Formats (ohne byte order marker (BOM)) kodiert sind und keinerlei Formatierungen enthalten.
- Verzichten Sie auf eigene Packages bei der Erstellung Ihrer Lösungen, d.h. **kein** `package`-Statement am Beginn Ihrer Java-Dateien.
- Quelltextkommentare können die Korrektoren beim Verständnis Ihrer Lösung (insbesondere bei inkorrekten Abgaben) unterstützen; sind aber nicht verpflichtend.
- Testen Sie Ihre Lösung bitte auf einem Rechner aus dem Informatik-Computer-Pool z.B. `gruenau6.informatik.hu-berlin.de` (siehe auch Praktikumsordnung: Referenzplattform)

Aufgabe 1 (Labyrinth) 10 Punkte

Gegeben sei ein Labyrinth der Dimension *n* x *n*, in dem mithilfe von Backtracking ein Weg von einem Start- zu einem Zielpunkt gefunden werden soll. Dabei kann das Labyrinth nach links oder nach unten durchlaufen werden. Wenn ein Zug nach links in eine Sackgasse führt, dann wird dieser zurückgenommen. Das wird solange wiederholt, bis ein Zug nach unten möglich ist. Das Beispiel in Abbildung 1.1 demonstriert das beschriebene Vorgehen.

Angefangen in der oberen rechten Ecke soll die untere linke Ecke erreicht werden. Zunächst wird das Labyrinth nach links durchlaufen, bis eine Sackgasse (S) erreicht wird. Ein Zug nach links wird zurückgenommen, bis ein Zug nach unten möglich ist. Da anschließend kein Zug nach links möglich ist, werden weitere Züge nach unten durchgeführt. Es folgt ein Zug nach links, bis das Ziel (x) erreicht wird.

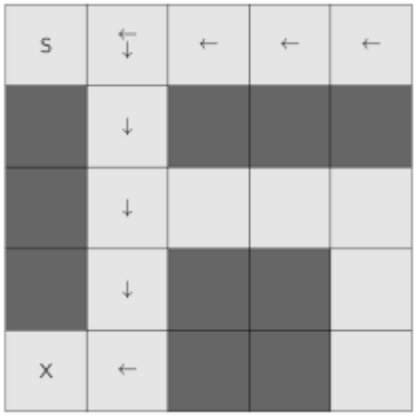

Abbildung 1.1: Beispiellabyrinth

Eine Lösung in Java kann mithilfe von Rekursion und Backtracking realisiert werden. Dazu wird jeweils ein Zug nach links bzw. nach unten vollzogen, bis

1. die Grenze des Labyrinths erreicht wurde,
2. der Zug in eine Sackgasse führt oder
3. das Ziel erreicht wurde.

Im 1. und 2. Fall wird (sofern möglich) ein Zug nach links vollzogen. Ansonsten wird der Zug „zurückgenommen“ und stattdessen (sofern möglich) ein Zug nach unten gemacht. Andernfalls wird ein weiterer Zug zurückgenommen. Im 3. Fall konnte ein Weg gefunden werden und das Labyrinth ist somit lösbar.

Ergänzen Sie in der vorgegebenen Java-Datei `MazeSolver.java`, die fehlenden Implementierungen für die Methoden `solve` und `draw`, welche das Lösen des Labyrinths (gemäß der oben genannten Regeln) und das Anzeigen der in `int[][] maze` repräsentierten Konfiguration, umsetzen sollen. Als Repräsentation des Labyrinths dient ein mehrdimensionales Array, wobei die erste Dimension die Zeilen und die zweite Dimension die Spalten des Labyrinths repräsentieren. Dabei gibt der Wert `2` den Start- und der Wert `3` den Zielpunkt an. Der Wert `1` repräsentiert ein begehbares Feld im Labyrinth und der Wert `0` steht für ein Hindernis. Das Labyrinth aus Abbildung 1.1 würde bspw. wie folgt repräsentiert:

```
int[][] maze = {
        {1, 1, 1, 1, 2},
        {0, 1, 0, 0, 0},
        {0, 1, 1, 1, 1},
        {0, 1, 0, 0, 1},
        {3, 1, 0, 0, 1}
};
```

Implementieren Sie die in Ihrer Klasse `MazeSolver` die von außen nutzbare Methode `public static boolean solve(int[][] maze, int row, int col)`. Die beiden Parameter vom Typ `int` repräsentieren dabei jeweils den aktuellen Index der Zeile bzw. Spalte. Der Rückgabewert vom Typ `boolean` gibt an, ob das Labyrinth lösbar ist, d. h. ob ausgehend vom Startpunkt der Zielpunkt erreicht werden kann. Implementieren Sie die Methode `solve` so, dass der schlussendlich gewählte Pfad durch das Labyrinth im Array visualisiert wird. Dazu sollen die Werte an den Stellen der jeweils gewählten Einträge durch den Wert `2` ersetzt werden.

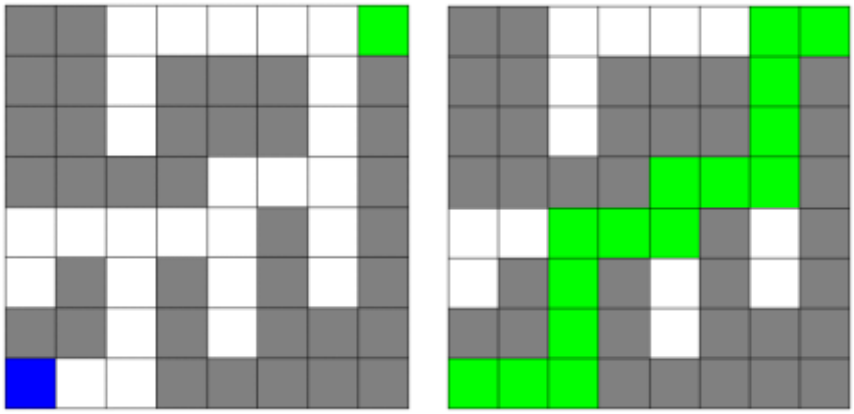

Abbildung 1.2: Labyrinth vor und nach Ausführung der Methode `solve()`

Bei der Implementierung der Methode `public static void draw(int[][] maze)` zum Anzeigen beachten Sie bitte folgende Konventionen:

- Passen Sie die Größe des Zeichenfeldes mithilfe der beiden Methoden `setXscale(double min, double max)` und `setYscale(double min, double max)` der Bibliothek `StdDraw` an.
- Der Startpunkt ist oben rechts, der Zielpunkt unten links.
- Jede in `maze` enthaltene Zahl wird als Quadrat in der folgenden Farbe (`0` → grau, `1` → weiß, `2` → grün, `3` → blau) dargestellt und mit einem schwarzen Rahmen umgeben.

Hinweise zur Bearbeitung:

- Sie können davon ausgehen, dass nur quadratische Labyrinthe übergeben werden.

- Bei der Entwicklung Ihrer Lösung kann es ggf. hilfreich sein den Zustand des Labyrinths (in der Entwicklungsphase) häufiger darzustellen, um so die Vorgehensweise Ihrer Lösung grafisch nachzuvollziehen.
- Testen Sie Ihre Lösungen mithilfe der Klasse `MazeTester`, welche Testfälle für sieben verschiedene lösbare oder unlösbare Labyrinthe beinhaltet.
- Zum Testen liegen `MazeSolver` und `MazeTester` im gleichen Verzeichnis und `gdp.stdlib.StdDraw` wird an diesem Platz erwartet.
- Laden Sie bitte genau die von Ihnen mit den Implementierungen ergänzte Datei `MazeSolver.java` in Moodle hoch.

Aufgabe 2 (Bitmanipulation) „Gray-Codes“ 10 Punkte

Gray-Codes finden Anwendung, wenn benachbarte Bitmuster einen Hamming-Abstand von 1 haben sollen. Eine Möglichkeit Gray-Codes zu erzeugen, ist die fortgesetzte Spiegelung und das Hinzufügen der Bits „0“ im oberen Teil und des Bits „1“ unteren Teil für eine um eins erhöhte Bitlänge b. Also:

b=1 (Start)

0
1

b=2 (Spiegeln, Auffüllen)

0	0
0	1
1	1
1	0

b=3 (Spiegeln, Auffüllen)

0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

…

b=31 (Spiegeln, Auffüllen)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
…																														
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Wird den Bitmustern, die man für b=31 erhält noch ein weiteres Bit „0“ vorangestellt, so ergibt sich gerade die Darstellung für eine der 2^{31} in Java durch **int** darstellbaren nicht negativen Zahlen. Wenn Bitmuster in der Tabelle b=31 dann von n=0, 1, …, 2^{31}-1

durchnummeriert, listet diese für jedes n den zugehörigen Gray-Code auf, der wiederum mit vorangestellten „0“-Bit als **int** in Java interpretiert werden kann. Es ist also implizit eine Funktion gegeben, die jeder positiven Integer in Java eine (in der Regel andere) positive Integer zuordnet.

Scheiben Sie eine Klasse **`Gray.java`**, die:

a. **[6 Punkte]:**
   Eine Funktion `public static int toGray(int n)` implementiert, die für nicht negative `n` den entsprechenden Gray-Code als nicht negative Integer liefert.

b. **[4 Punkte]:**
   Eine Funktion `public static int fromGray(int g)` implementiert, die für nicht negative Integer g, die einen Gray-Code darstellen sollen, den ursprünglichen Wert `n`, also gerade die Umkehrfunktion berechnet.

Hinweise:

- Die beiden Funktionen müssen im Wertebereich `{0, …, Integer.MAX_VALUE}` korrekt funktionieren.
- Die Funktionen müssen `public static` sein, damit sie auch von anderen Java-Klassen (z.B. der bereitgestellten Testklasse `GrayTester.java`) verwendet werden können.
- Die Abbildung ist bijektiv.
- `toGray(4) == 6`
- **Testfälle:** `GrayTester.java (javac Gray*.java && java GrayTester)`
  1. `toGray( 0 ) == 0`?
  2. `fromGray( 0 ) == 0`?
  3. `toGray( n ) ^ toGray( n+1 )`
     liefert einen Bit-Count von „1“ (Hamming-Abstand=1)
  4. `fromGray( toGray( n ) ) == n` ?