Aufgabenblatt 5

Kompetenzstufe 1 & Kompetenzstufe 2

Allgemeine Informationen zum Aufgabenblatt:

- Die Abgabe erfolgt in TUWEL. Bitte laden Sie Ihr IntelliJ-Projekt bis spätestens **Donnerstag, 05.01.2023 20:00 Uhr** in TUWEL hoch.
- Zusätzlich müssen Sie in TUWEL ankreuzen, welche Aufgaben Sie gelöst haben.
- Ihre Programme müssen kompilierbar und korrekt ausführbar sein.
- Ändern Sie bitte **nicht** die **Dateinamen** und die **vorhandene Ordnerstruktur**.
- Verwenden Sie, falls nicht anders angegeben, für alle Ausgaben `System.out.println()` bzw. `System.out.print()`.
- Verwenden Sie für die Lösung der Aufgaben keine Aufrufe (Klassen) aus der Java-API, außer diese sind ausdrücklich erlaubt.
- Erlaubt sind die Klassen `String`, `Math`, `Integer` und `CodeDraw` oder Klassen, die in den Hinweisen zu den einzelnen Aufgaben aufscheinen.
- Bitte beachten Sie die Vorbedingungen! Sie dürfen sich darauf verlassen, dass alle Aufrufe die genannten Vorbedingungen erfüllen. Sie müssen diese nicht in den Methoden überprüfen.

In diesem Aufgabenblatt werden folgende Themen behandelt:

- Ein- und zweidimensionale Arrays
- Rekursion
- Grafische Ausgabe
- Zweidimensionale Arrays und Bilder

Aufgabe 1 (1 Punkt)

Implementieren Sie folgende Aufgabenstellung:

a) Implementieren Sie eine Methode `shiftLines`:

```
void shiftLines(int[][] workArray)
```

Diese Methode baut ein ganzzahliges zweidimensionales Array `workArray` so um, dass die Zeile, deren letztes Element am kleinsten ist, in die erste Zeile verschoben wird. Die erste Zeile wird dann an jene Stelle verschoben, wo zuvor die Zeile mit dem kleinsten Element an der letzten Stelle gewesen ist. Sollte das kleinste Element bei mehreren Zeilen an der letzten Stelle vorkommen, so wird nur jene Zeile verschoben, die den kleinsten Index aufweist. Dafür wird das Array `workArray` umgebaut und kein neues Array erstellt. Sie dürfen aber innerhalb der Methode eine temporäre Array-Variable anlegen.

Vorbedingungen: `workArray != null`, `workArray.length > 0` und für alle gültigen `i` gilt `workArray[i].length > 0`.

Beispiele:

Aufruf	Ergebnis

`shiftLines(`
`new int[][]{`
`{1,3,2},`
`{6,2,5},`
`{0,7,9}})`
	`1 3 2`
	`6 2 5`
	`0 7 9`

`shiftLines(`
`new int[][]{`
`{1,5,6,7},`
`{1,9,6},`
`{4,3},`
`{6,3,0,6,9},`
`{6,4,3}})`
	`4 3`
	`1 9 6`
	`1 5 6 7`
	`6 3 0 6 9`
	`6 4 3`

`shiftLines(`
`new int[][]{`
`{7,3,6},`
`{5},`
`{9,1},`
`{3,2,4,1},`
`{0}})`
	`0`
	`5`
	`9 1`
	`3 2 4 1`
	`7 3 6`

b) Implementieren Sie eine Methode `addEntries`:

```
void addEntries(int[][] workArray)
```

Diese Methode fügt in jeder Zeile von `workArray` so viele Elemente hinzu, wie die größte Zahl in dieser Zeile angibt. Die hinzugefügten Elemente werden mit der größten Zahl dieser Zeile befüllt. Dafür wird das Array `workArray` umgebaut und kein neues Array erstellt. Sie dürfen aber innerhalb der Methode ein temporäres Hilfs-Array anlegen.

Vorbedingungen: `workArray != null`, `workArray.length > 0` und für alle gültigen `i` gilt `workArray[i].length > 0`.

Beispiele:

Aufruf	Ergebnis

`addEntries(`
`new int[][]{`
`{1},`
`{1,2},`
`{1,2,3}})`
	`1 1`
	`1 2 2 2`
	`1 2 3 3 3 3`

`addEntries(`
`new int[][]{`
`{3,4,2},`
`{1,3,2},`
`{5,0,1}})`
	`3 4 2 4 4 4 4`
	`1 3 2 3 3 3`
	`5 0 1 5 5 5 5 5`

`addEntries(`
`new int[][]{`
`{1,2},`
`{1,2,4,3},`
`{6},`
`{1,2,5,3,4},`
`{1},`
`{3}})`
	`1 2 2 2`
	`1 2 4 3 4 4 4 4`
	`6 6 6 6 6 6 6`
	`1 2 5 3 4 5 5 5 5 5`
	`1 1`
	`3 3 3 3`

Aufgabe 2 (1 Punkt)

Implementieren Sie folgende Aufgabenstellung:

- Implementieren Sie eine Methode `genCircleFilter`:

  ```
  double[][] genCircleFilter(int n, double radius)
  ```

  Die Methode erzeugt ein zweidimensionales Array (Filter) der Größe `n` × `n`, das nur die Zahlen `0` und `1` enthält. Jedes Element, dessen Abstand zum Mittelpunkt des Arrays kleiner als `radius` ist, wird auf `1` gesetzt, die übrigen Elemente auf `0`. Es wird daher ein kreisförmiges Muster um die Mitte des Arrays erzeugt. Der Abstand zum Mittelpunkt (z.B. liegt der Mittelpunkt bei einem `3` × `3` Array an der Stelle `[1][1]`) lässt sich dabei über die euklidische Distanz $\sqrt{\Delta x^2 + \Delta y^2}$ berechnen, wobei Δx und Δy dem Abstand der x-Koordinate (entspricht dem Spaltenindex in dem Array) bzw. y-Koordinate (entspricht dem Zeilenindex in dem Array) zum Mittelpunkt entsprechen. Überprüfen Sie in der Methode auch, ob der Eingabewert `n` ungerade und größer gleich 1 ist, ansonsten geben Sie den Wert `null` zurück.

  Beispiele:

  `genCircleFilter(3, 1.2)` erzeugt →

  ```
  0,00 1,00 0,00
  1,00 1,00 1,00
  0,00 1,00 0,00
  ```

  Der Wert an der Stelle `[0][0]` wird auf 0 gesetzt, da sein Abstand zum Mittelpunkt $\sqrt{1^2 + 1^2}$ größer als 1.2 ist. Der Wert an der Stelle `[0][1]` wird auf 1 gesetzt, da sein Abstand zum Mittelpunkt $\sqrt{0^2 + 1^2}$ kleiner als 1.2 ist.

  `genCircleFilter(7, 2.5)` erzeugt →

  ```
  0,00 0,00 0,00 0,00 0,00 0,00 0,00
  0,00 0,00 1,00 1,00 1,00 0,00 0,00
  0,00 1,00 1,00 1,00 1,00 1,00 0,00
  0,00 1,00 1,00 1,00 1,00 1,00 0,00
  0,00 1,00 1,00 1,00 1,00 1,00 0,00
  0,00 0,00 1,00 1,00 1,00 0,00 0,00
  0,00 0,00 0,00 0,00 0,00 0,00 0,00
  ```

- Implementieren Sie eine Methode `applyFilter`:

  ```
  double[][] applyFilter(double[][] workArray, double[][] filterArray)
  ```

  Diese Methode wendet einen Filter `filterArray` auf ein gegebenes rechteckiges Array `workArray` an (berechnet die Kreuzkorrelation). Dazu erzeugt die Methode ein neues Array, das dieselbe Größe wie `workArray` hat. Dabei beschreibt `filterArray` ein Muster um einen Mittelpunkt herum, das an allen Positionen über `workArray` gelegt wird, an denen `filterArray` vollständig hineinpasst. Bei jedem Überlagern kann der Wert des Rückgabe-Arrays am Mittelpunkt von `filterArray` folgendermaßen berechnet werden: Für jeden

überlagerten Punkt wird zuerst das Produkt der entsprechenden Werte in `filterArray` und `workArray` gebildet. Der Wert im Rückgabe-Array ist dann die Summe dieser Produkte. Steht `filterArray` bei der Anwendung über den Rand hinaus, dann wird keine Berechnung durchgeführt und an der entsprechenden Stelle die Zahl 0 eingetragen.

Vorbedingungen: `workArray != null`, `workArray.length > 0`, für alle gültigen `i` gilt, dass `workArray[i].length` demselben konstanten Wert größer 0 entspricht; `filterArray != null`, `filterArray.length > 0` und ungerade, für alle gültigen `i` gilt, dass `filterArray[i].length` demselben konstanten und ungeraden Wert größer 0 entspricht.

Ist beispielsweise das `workArray` gegeben als

```
0 1 2  3
4 5 6  7
8 9 10 11
```

und das `filterArray` gegeben als

```
1 0 0
1 2 0
0 0 3
```

ergibt sich als Ausgabewert an der Stelle `[1][1]` der Wert $0 \cdot 1 + 1 \cdot 0 + 2 \cdot 0 + 4 \cdot 1 + 5 \cdot 2 + 6 \cdot 0 + 8 \cdot 0 + 9 \cdot 0 + 10 \cdot 3 = 44$. Diese Berechnung wird für alle gültigen Positionen durchgeführt. Das Ergebnis-Array würde in diesem Fall folgendermaßen aussehen:

```
0 0  0  0
0 44 51 0
0 0  0  0
```

Die gesamte Filteroperation ist zusätzlich noch in Abbildung 1 veranschaulicht.

- Testen Sie Ihre Methoden mit den in `main` zur Verfügung gestellten Aufrufen. Wenden Sie zusätzlich folgenden Filter auf das in `main` vorgegebene Array `myArray4` an:

```
0,00 0,00 0,00
0,00 0,00 0,00
0,00 0,50 0,00
```

Geben Sie das Ergebnis auf der Konsole aus. Bei richtiger Implementierung müssen die Werte im Array halbiert und um eine Stelle nach oben verschoben worden sein.

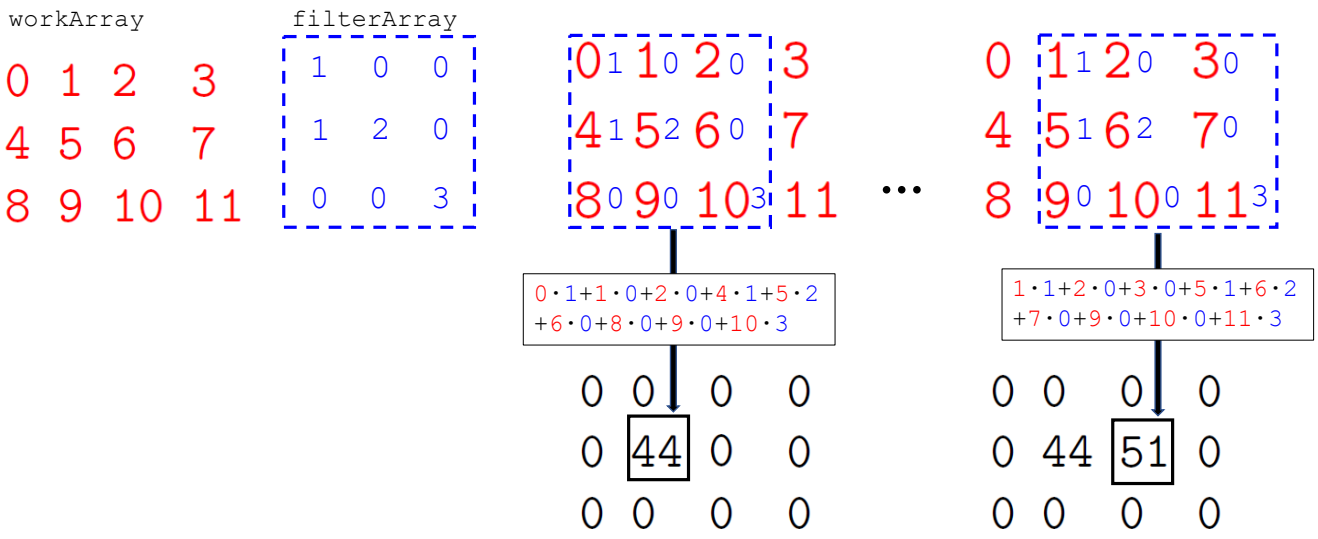


Abbildung 1: Veranschaulichung der einzelnen Schritte der Filteroperation für ein gegebenes `workArray` und `filterArray`. Um das Ergebnis für eine bestimmte Stelle zu berechnen, wird das `filterArray` mittig über die entsprechende Stelle im `workArray` gelegt. Das Ergebnis wird zuerst für die Stelle `[1][1]` berechnet, da hier das `filterArray` vollständig in das `workArray` hineinpasst (bei den Stellen in der ersten Zeile und ersten Spalte von `workArray` würde das `filterArray` über den Rand des `workArray` hinausgehen, deswegen kann dort kein Ergebnis berechnet werden). Für das Ergebnis an der Stelle `[1][1]` werden nun die korrespondierenden Elemente im `workArray` und `filterArray` multipliziert und aufsummiert (Ergebnis: 44). Im nächsten Schritt wird das `filterArray` um eine Position nach rechts auf die Stelle `[1][2]` verschoben und die gleiche Berechnung durchgeführt (Ergebnis: 51). Für die restlichen Stellen wird keine Berechnung durchgeführt, da das `filterArray` dort nicht vollständig in das `workArray` hineinpasst (Ergebnis: 0).

Aufgabe 3 (2 Punkte)

Bei dieser Aufgabe soll ähnlich wie bei Aufgabe 2 eine lokale Operation auf alle Elemente eines 2D-Arrays angewendet werden. In diesem Fall stellen die Elemente des 2D-Arrays die Pixelwerte eines digitalen Bildes dar. Konkret geht es darum, in solch einem Bild die *Waldo* Figur zu finden, angelehnt an die bekannten *Where's Waldo?* Kinderbücher[1]. Einige Teile des Programms (z.B. Laden und Konvertieren der Bilder) wurden schon implementiert.

Implementieren Sie dazu folgende Aufgabenstellung:

- Implementieren Sie eine Methode `detectWaldo`:

  ```
  void detectWaldo(CodeDraw myDrawObj, Image img, Image template)
  ```

  Die Methode übernimmt ein Bild `img` und ein Template `template` (Waldo), sucht das Template im Bild und zeigt die aktuell gefundene Position als Bounding Box an. Analog zur Aufgabe 2 wandert das `template` über das Bild `img`, wobei an jeder Stelle eine lokale Operation durchgeführt wird. Da wir in diesem Fall die Ähnlichkeit des Templates zum lokalen Bildausschnitt berechnen möchten, berechnen wir nicht wie in Aufgabe 2 die Kreuzkorrelation, sondern die *Summe der absoluten Differenzen* (*Sum of Absolute Differences, SAD*): alle korrespondierenden Pixelwerte werden subtrahiert und die Absolutbeträge all dieser Differenzen werden aufsummiert. Diese Vorgehensweise liefert uns für jeden Bildpunkt ein Maß der *Unähnlichkeit*, und wir können Waldo dort finden, wo dieses Maß am geringsten ist (Hinweis: dieses Maß ist in unserem Fall an der korrekten Waldo-Stelle nicht 0, da Template- und Bildinhalt nicht 100%-ig ident sind).

  Zu Beginn müssen die Farbbilder `img` und `template` in zweidimensionale Arrays umgewandelt werden, in denen jeder Eintrag den Grauwert des jeweiligen Pixels darstellt. Hierfür können Sie die bereits vorhandene Methode `convertImg2Array` verwenden.

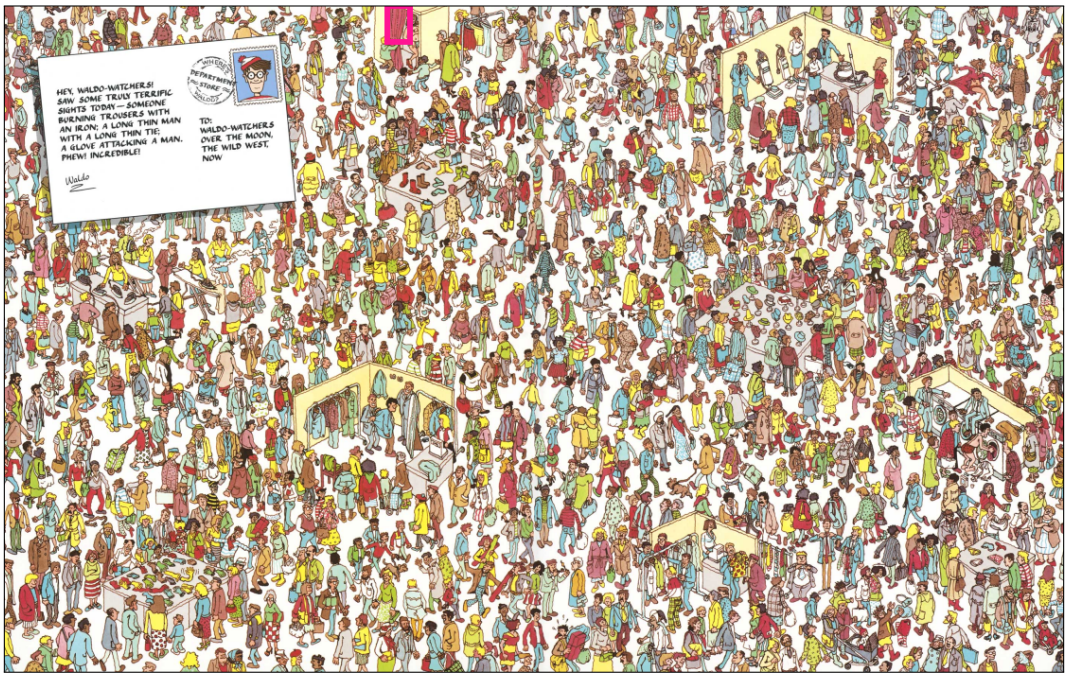
  Analog zu Aufgabe 2 werden nur Positionen ausgewertet, bei denen das Template nicht über den Rand des Bildes hinausgeht. Wurde eine Stelle mit minimaler Unähnlichkeit gefunden, wird die entsprechende Bounding Box im Bild angezeigt. Der von der Bounding Box angezeigte Bildausschnitt soll dabei dieselbe Größe wie das `templateArray` haben. Verwenden Sie für die Bounding Box die Farbe `Palette.DEEP_PINK` und eine Linienbreite von `6`. Es soll immer nur die aktuell gefundene Position von Waldo angezeigt werden (mit minimaler *SAD*). Jedes Mal, wenn eine neue Position gefunden wurde, soll auch eine Pause von 300ms eingelegt werden. Verwenden Sie dafür die Methode `myDrawObj.show(long waitMilliseconds)`. Bei einer richtigen Implementierung werden üblicherweise zu Beginn verschiedene falsche Positionen angezeigt, bis der richtige Waldo gefunden wird.

  Vorbedingungen: `myDrawObj != null`, `img != null`, `template != null`.

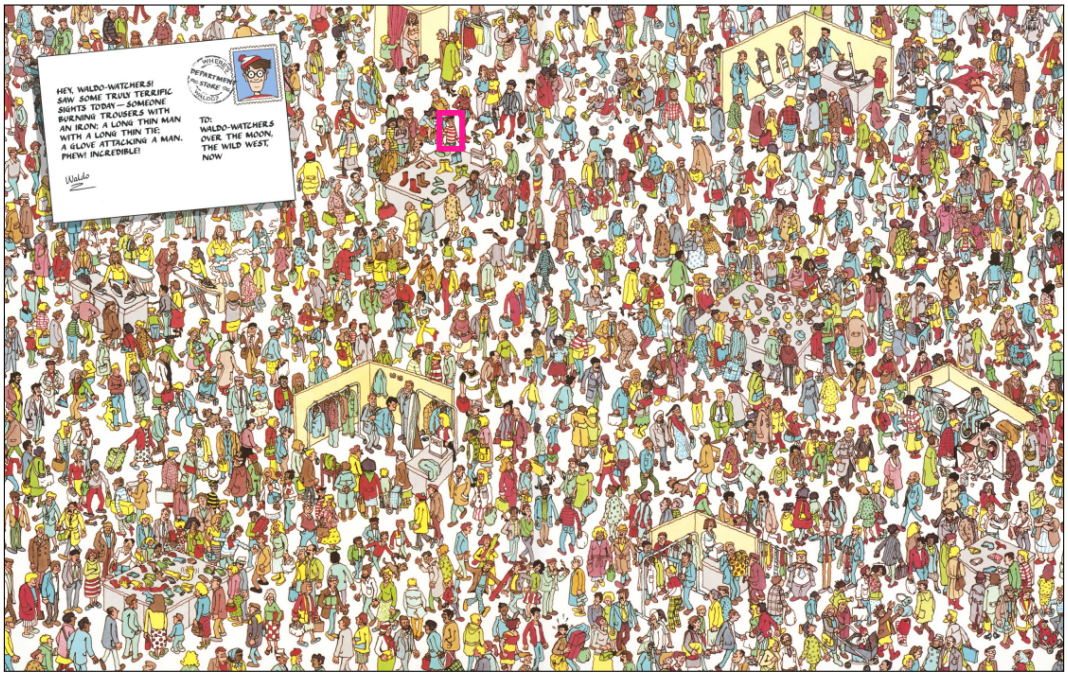
- Sie können ihre Methode zunächst mit dem Eingabebild `waldo1.png` und dem Template `template1.png` testen (Hinweis: Die Bilddateien sind als Links hinterlegt, damit diese nicht bei jedem IntelliJ-Projekt in TUWEL hochgeladen werden müssen). In diesem Fall muss ein Endergebnis wie in Abbildung 2b herauskommen (die minimale SAD beträgt in diesem Fall). Abbildung 2a zeigt ein mögliches Zwischenergebnis, bei dem noch nicht die korrekte Position gefunden wurde.

[1] `https://en.wikipedia.org/wiki/Where%27s_Wally%3F`

- Testen Sie Ihre Lösung auch mit den Bildern `waldo2.png` und `waldo3.png` sowie den zugehörigen Templates `template2.png` und `template3.png` und überprüfen Sie, ob Waldo korrekt gefunden wurde. Dazu bitte bei den entsprechenden Links in `main` die Kommentare entfernen.



(a)



(b)

Abbildung 2: a) Zwischenergebnis mit falscher Detektion (am oberen Bildrand) und b) korrektes Endergebnis mit markiertem Waldo.

Aufgabe 4 (2 Punkte)

Implementieren Sie folgende Aufgabenstellung:

- Bei dieser Aufgabe haben Sie ein Labyrinth in Form eines zweidimensionalen Arrays gegeben. In diesem Labyrinth soll ein Roboter von einem beliebigen Startpunkt aus versuchen, einen Ausgang zu finden. Es sind in `main` bereits die Datenstrukturen und entsprechenden Aufrufe der Methoden vorgegeben, die nachfolgend beschrieben werden.

  In `main` finden Sie eine Variable `mazeType`, die für die Auswahl eines der drei vorgegebenen Labyrinthe verwendet wird.

  Ein Labyrinth wird im zweidimensionalen Array `myMaze` abgelegt. Dieses Array hat 15 Zeilen und 15 Spalten und ist vom Datentyp `char`. Folgende Zeichen wurden für die Codierung der Felder verwendet:

  1. `' '` ... entspricht einem leeren Feld und kann bei der Wegsuche verwendet werden.
  2. `'*'` ... entspricht einer Wand und wird als Hindernis interpretiert.
  3. `'S'` ... entspricht dem Startpunkt, wo die Suche beginnt.
  4. `'E'` ... entspricht dem Ausgang, den der Roboter finden soll.

  Das zweidimensionale Array `direction` beschreibt die Suchreihenfolge innerhalb des Labyrinths. Dieses Array hat vier Zeilen und jede Zeile entspricht einer von vier Himmelsrichtungen (North (N), East (E), South (S), West (W)). Diese Richtungen können innerhalb von `direction` in unterschiedlicher Reihenfolge vorkommen. Die erste Spalte einer Zeile entspricht der Veränderung in y-Richtung (Zeile in `myMaze`) und die zweite Spalte der Veränderung in x-Richtung (Spalte in `myMaze`). Steht zum Beispiel in einer Zeile von `direction` die Kombination `[0,-1]`, dann entspricht das einer Suche nach links (Westen), da es einer Änderung der y-Richtung um 0 entspricht und einer Änderung der x-Richtung um -1. Ist `direction` beispielweise deklariert als

  ```
  int[][] direction = new int[][]{{0,1},{-1,0},{0,-1},{1,0}};,
  ```

  dann entspricht das folgender Reihenfolge von Himmelsrichtungen: E, N, W, S

  In `main` ist auch der Aufruf der verschiedenen Methoden ersichtlich. Es wird zuerst der Startpunkt ermittelt (`getStartPoint`), dann wird die Suche aufgerufen (`existsPathToExit`) und zuletzt wird das Ergebnis auf der Konsole ausgegeben (`printMaze`). Die drei Methoden werden nachfolgend im Detail beschrieben:

(!) Gilt für die zu implementierenden Methoden: Sie dürfen keine Klassenvariablen verwenden. Der vorgegebene Methodenkopf darf nicht erweitert oder geändert werden.

- Implementieren Sie eine Methode `getStartPoint`:

  ```
  int[] getStartPoint(char[][] maze)
  ```

  Diese Methode sucht innerhalb des zweidimensionales Arrays `maze` den Startpunkt, der mit einem `'S'` gekennzeichnet wird. Die Koordinaten des gefundenen Startpunkts werden in einem eindimensionalen Array gespeichert, das innerhalb dieser Methode erzeugt wird. Am Index `0` des erzeugten Arrays wird die Zeile des Startpunktes gespeichert und am Index `1` die entsprechende Spalte. Das Array mit den Koordinaten des Startpunktes wird zurückgegeben. Vorbedingung: `maze != null`.

- Implementieren Sie die rekursive Methode `existsPathToExit`:

```
boolean existsPathToExit(char[][] maze,
  int row, int col, int[][] direction)
```

  Diese Methode überprüft das zweidimensionale Array `maze`, ob ein Ausgang für den Roboter vorhanden ist. Die Methode bekommt zusätzlich als Parameter den Wert `row` für die Zeile und den Wert `col` für die Spalte übergeben. Beide Werte beschreiben die Position, auf der die Suche innerhalb des Labyrinths (Array) aktuell stattfindet. Der letzte Parameter `direction` ist ein zweidimensionales Array für die Suchreihenfolge innerhalb des Labyrinths. Beim ersten Aufruf der Methode steht in `row` und `col` der Startpunkt des Roboters. Sie müssen rekursiv in alle vier Richtungen, laut Reihenfolge in `direction`, nach dem Ausgang suchen. Dazu müssen Sie die Methode rekursiv mit immer neuen `row`- und `col`-Werten aufrufen. Dabei müssen Sie berücksichtigen, dass Sie nur Felder betreten dürfen, die noch frei (Zeichen `' '`) sind und keiner Wand entsprechen bzw. schon betreten wurden. Während der Suche markieren Sie innerhalb des Arrays `maze` die betretenen Felder mit dem Zeichen `'V'` (Visited). Sollte der Ausgang gefunden worden sein, dann markieren Sie die Felder innerhalb des Arrays `maze`, die dem Pfad zum Ausgang entsprechen, mit dem Zeichen `'U'` (Used). Die Suche ist beendet, wenn der Ausgang gefunden wurde, oder alle vom Startpunkt erreichbaren Felder besucht wurden. Wurde der Ausgang gefunden, dann wird `true` zurückgegeben, ansonsten `false`. Vorbedingungen: `maze != null` und `direction != null`.

- Implementieren Sie die Methode `printMaze`:

```
void printMaze(char[][] maze)
```

  Diese Methode gibt das zweidimensionale Array `maze` auf der Konsole aus.

- In den nachfolgenden Beispielen werden verschiedene Labyrinthe und Ergebnisse der Ausgangssuche dargestellt. In Abbildung a) sehen Sie die Darstellung eines Labyrinths (`mazeType = 0`), bevor die Suche stattgefunden hat. Wird in diesem Labyrinth gesucht, dann wird das Ergebnis Abbildung b) erzielt. Der Ausgang ist von Wänden eingesperrt und somit nicht erreichbar. Aus diesem Grund sind nur die `'V'` (Visited) Markierungen auf allen erreichbaren und besuchten Feldern zu sehen. Wird die Wand zum Ausgang entfernt (entspricht `mazeType = 1`) und die Suche erneut ausgeführt, dann erhalten wir das Ergebnis in Abbildung c), wenn für `direction` die Version (E,N,W,S) verwendet wird. Der Ausgang ist erreichbar und der finale Pfad wird durch die `'U'` (Used) Markierungen gekennzeichnet. Alle anderen besuchten Felder, die nicht dem Pfad zum Ausgang laut Suchreihenfolge entsprechen, werden mit den `'V'` (Visited) Markierungen dargestellt. An den unterschiedlichen Beispielen ist ersichtlich, dass bei der Suche Wege beschritten werden können, die in Sackgassen führen. Die Abbildungen d) bis f) zeigen die unterschiedlichen Ergebnisse, wenn die Suche mit den Suchreihenfolgen (N,W,S,E), (W,S,E,N) und (S,E,N,W) für `direction` und `mazeType = 1` durchgeführt wird.

a) Leeres Labyrinth

```
***************
*         *   E
***** *   *   *
*   * *********
*             *
***********   *
*             *
* ***  ****   *
*   *     *   *
***** *****   *
*         *   *
* ********** **
*         *   *
*  S *        *
***************
```

b) Kein Pfad zum Ausgang

```
***************
*VVVVVVVVV*   E
*****V*VVVV*  *
*VVV*V*********
*VVVVVVVVVVVVV*
***********VVV*
*VVVVVVVVVVVVV*
*V***VV****VVV*
*VVV*VVVVV*VVV*
*****V*****VVV*
*VVVVVVVV*VVVV*
*V**********V**
*VVVVVVVV*VVVV*
*VVSV*VVVVVVVV*
***************
```

c) Suche mit Reihenfolge E,N,W,S

```
***************
*         *  UE
***** *    * U*
*   * *******U*
*            U*
***********  U*
*            U*
* ***  ****  U*
*   *     *  U*
***** *****  U*
*         *  UU*
* **********U**
*   UUUUU*  UU*
*  SU*  UUUUUU*
***************
```

d) Suche mit Reihenfolge N,W,S,E

```
***************
*VVVVVVVVV*VVUE
*****V*VVVV*VU*
*VVV*V*******U*
*VVVVVVVVVVUUU*
***********UUU*
*VVVVUUUUUUUUU*
*V***UV****UUU*
*VVV*UVVVV*UUU*
*****U*****UU *
*UUUUU   *VUU *
*U********** **
*UUU     *    *
*  S *        *
***************
```

e) Suche mit Reihenfolge W,S,E,N

```
***************
*VVVVVVVVV*VUUE
*****V*VVVV*UU*
*VVV*V*******U*
*VVVVVVVVVVVVU*
***********UUU*
*VVVVVVVVVVUUU*
*V***VV****UUU*
*VVV*VVVVV*UUU*
*****V*****UUU*
*VVVVVVVV*VUUV*
*V**********U**
*UUUUUU  *VVUU*
*UUSV*UUUUUUUU*
***************
```

f) Suche mit Reihenfolge S,E,N,W

```
***************
*         *  UE
***** *    * U*
*   * *******U*
*            U*
***********  U*
*            U*
* ***  ****  U*
*   *     *  U*
***** *****  U*
*         *  UU*
* **********U**
*   UUU  *  UU*
*  SU*UUUUUUUU*
***************
```