Aufgabenblatt 5

Kompetenzstufe 1 & Kompetenzstufe 2

Allgemeine Informationen zum Aufgabenblatt:

- Die Abgabe erfolgt in TUWEL. Bitte laden Sie Ihr IntelliJ-Projekt bis spätestens **Freitag, 08.01.2021 15:00 Uhr** in TUWEL hoch.
- Zusätzlich müssen Sie in TUWEL ankreuzen, welche Aufgaben Sie gelöst haben.
- Ihre Programme müssen kompilierbar und 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 `StdDraw` 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:

- Implementieren Sie eine Methode `genFilledArray`:

  ```
  int[][] genFilledArray(int n)
  ```

  Die Methode erzeugt ein zweidimensionales Array der Größe n×n und befüllt dieses mit Zahlen, wie in den nachfolgenden Beispielen gezeigt. Es wird links oben (0,0) mit 1 begonnen und in jeder weiteren Gegendiagonalen die Zahl um 1 vergrößert. Nach der Zahl `n` wird mit jeder weiteren Gegendiagonalen die Zahl wieder um 1 verringert, sodass es rechts unten wieder mit der Zahl 1 endet. Anschließend wird das neu erzeugte Array zurückgegeben.

  Vorbedingung: `n > 0`.

  Beispiele:

  `genFilledArray(2)` erzeugt →

  ```
  1 2
  2 1
  ```

  `genFilledArray(4)` erzeugt →

  ```
  1 2 3 4
  2 3 4 3
  3 4 3 2
  4 3 2 1
  ```

  `genFilledArray(7)` erzeugt →

  ```
  1 2 3 4 5 6 7
  2 3 4 5 6 7 6
  3 4 5 6 7 6 5
  4 5 6 7 6 5 4
  5 6 7 6 5 4 3
  6 7 6 5 4 3 2
  7 6 5 4 3 2 1
  ```

- Implementieren Sie eine Methode `shiftLinesInArray`:

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

  Diese Methode baut ein ganzzahliges zweidimensionales Array `workArray` so um, sodass alle Zeilen innerhalb des Arrays um eine Zeile nach unten verschoben werden. Die unterste Zeile wird ganz oben im Array wieder eingefügt. Dafür wird das Array `workArray` umgebaut und kein neues Array erstellt.

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

  Beispiele:

Aufruf	Ergebnis

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

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

- Implementieren Sie eine Methode `extendArray`:

  ```
  int[][] extendArray(int[][] inputArray)
  ```

  Diese Methode erstellt ein ganzzahliges zweidimensionales Array, bei dem jede Zeile die gleiche Länge aufweist. Die Länge der Zeilen wird durch die längste Zeile von `inputArray` bestimmt. Das Array `inputArray` kann unterschiedliche Zeilenlängen aufweisen. Die einzelnen Zeilen von `inputArray` werden dabei abwechselnd links und rechts mit Nullen aufgefüllt, sodass alle Zeilen des neuen Arrays gleich viele Einträge haben. Die erste Zeile beginnt mit dem Auffüllen der Nullen von links, die zweite Zeile von rechts.

  Vorbedingungen: `inputArray != null` und `inputArray.length > 0`, dann gilt auch für alle gültigen `i`, dass `inputArray[i].length > 0`.

  Beispiele:

Aufruf	Ergebnis

`extendArray(`
`new int[][]{`
`{4},`
`{1, 2, 3},`
`{5, 6},`
`{7, 8, 9, 1}})`
	`0 0 0 4`
	`1 2 3 0`
	`0 0 5 6`
	`7 8 9 1`

`extendArray(`
`new int[][]{`
`{1, 0, 1, 1, 0, 0, 0, 0},`
`{0, 1, 1, 1, 1, 1},`
`{1, 1},`
`{1, 0, 0, 0},`
`{1, 1, 0, 1},`
`{1},`
`{1}})`
	`1 0 1 1 0 0 0 0`
	`0 1 1 1 1 1 0 0`
	`0 0 0 0 0 0 1 1`
	`1 0 0 0 0 0 0 0`
	`0 0 0 0 1 1 0 1`
	`1 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 1`

`extendArray(`
`new int[][]{`
`{1, 3, 2},`
`{5, 1},`
`{6, 8, 5, 4},`
`{9, 4, 1, 9, 2},`
`{1, 8, 7, 5, 3, 2, 5},`
`{3}})`
	`0 0 0 0 1 3 2`
	`5 1 0 0 0 0 0`
	`0 0 0 6 8 5 4`
	`9 4 1 9 2 0 0`
	`1 8 7 5 3 2 5`
	`3 0 0 0 0 0 0`

- Implementieren Sie eine Methode `reformatArray`:

  `int[] reformatArray(int[][] inputArray)`

  Diese Methode interpretiert jede Zeile von `inputArray` als Binärzahl und erstellt ein neues eindimensionales Array, das jede dieser Binärzahlen als Dezimalzahl beinhalten soll. Das Array mit den Dezimalzahlen wird anschließend zurückgegeben. Jede Zeile von `inputArray` kann von hinten nach vorne gelesen werden und dabei kann die Wertigkeit jeder Stelle ermittelt werden. Das letzte Element in jeder Zeile von `inputArray` hat die Wertigkeit 2^0, das vorletzte Element jeder Zeile die Wertigkeit 2^1, usw. Nach diesem Schema wird jede Zeile durch Aufsummieren der einzelnen Wertigkeiten in eine Dezimalzahl umgewandelt. Die so entstandenen Dezimalzahlen werden im neuen Array der Reihe nach, beginnend beim Index 0, abgelegt.

  Vorbedingungen: `inputArray != null`, `inputArray.length > 0`, dann gilt für alle gültigen `i`, dass `inputArray[i].length > 0` $\wedge$ `inputArray[i].length < 32` ist. Alle Zahlen in `inputArray` sind Nullen oder Einsen.

  Beispiele:

  ```
  reformatArray(new int[][]{
  {1,0,0},
  {0,1,1}})
  ```
  erzeugt $\rightarrow$

  Zeile 1: `{1,0,0}` $\rightarrow 2^2 * 1 + 2^1 * 0 + 2^0 * 0 = 4$
  Zeile 2: `{0,1,1}` $\rightarrow 2^2 * 0 + 2^1 * 1 + 2^0 * 1 = 3$

  ```
  4 3
  ```

  ```
  reformatArray(new int[][]{
  {1, 0, 1, 1},
  {0, 1, 1},
  {1, 1, 0, 0, 0},
  {0, 0, 0, 1, 0},
  {1, 0},
  {1, 1, 1, 1, 1}})
  ```
  erzeugt $\rightarrow$

  ```
  11 3 24 2 2 31
  ```

  ```
  reformatArray(new int[][]{
  {1,0,1,1,0,0,0,0},
  {0,1,1,1,1,1,0,0},
  {0,0,0,0,0,0,1,1},
  {1,0,0,0,0,0,0,0},
  {0,0,0,0,1,1,0,1},
  {1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,1}})
  ```
  erzeugt $\rightarrow$

  ```
  176 124 3 128 13 128 1
  ```

Aufgabe 2 (1 Punkt)

Implementieren Sie folgende Aufgabenstellung:

- Implementieren Sie eine Methode `genMeanFilter`:

  ```
  double[][] genMeanFilter(int n)
  ```

  Die Methode erzeugt einen Mittelwertfilter[1] der Größe `n` × `n`. Jedes Element eines Mittelwertfilters entspricht dem Kehrwert der Gesamtanzahl der Array-Elemente. Bei `n==3` wird das Array somit beispielsweise mit dem Wert 1/9 befüllt. Überprüfen Sie in der Methode auch, ob der Eingabewert `n` ungerade und größer gleich 1 ist, ansonsten geben Sie den Wert `null` zurück.

  Beispiele:

  `genMeanFilter(3)` erzeugt →

  ```
  0,11 0,11 0,11
  0,11 0,11 0,11
  0,11 0,11 0,11
  ```

  `genMeanFilter(5)` erzeugt →

  ```
  0,04 0,04 0,04 0,04 0,04
  0,04 0,04 0,04 0,04 0,04
  0,04 0,04 0,04 0,04 0,04
  0,04 0,04 0,04 0,04 0,04
  0,04 0,04 0,04 0,04 0,04
  ```

- Implementieren Sie eine Methode `applyFilter`:

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

  Diese Methode wendet einen Filter `filterArray` auf ein gegebenes rechteckiges Array `workArray` an und erzeugt ein neues Array, das dieselbe Größe wie `workArray` hat. Dabei beschreibt `filterArray` ein Muster um einen Mittelpunkt herum, das an alle Positionen über `workArray` gelegt wird, an denen `filterArray` vollständig hineinpasst. Bei jedem Überlagern kann der Wert des Rückgabearrays 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 Rückgabearray 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` dem selben konstanten Wert größer 0 entspricht; `filterArray != null`, `filterArray.length > 0` und ungerade, für alle gültigen `i` gilt, dass `filterArray[i].length` dem selben konstanten und ungeraden Wert größer 0 entspricht.

[1] `https://de.wikipedia.org/wiki/Faltungsmatrix#Beispiele`

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 `myArray2` an:

```
0 1 0
0 0 0
0 0 0
```

Geben Sie das Ergebnis in der Konsole aus. Bei richtiger Implementierung müssen die Werte im Array um eine Stelle nach unten verschoben worden sein.

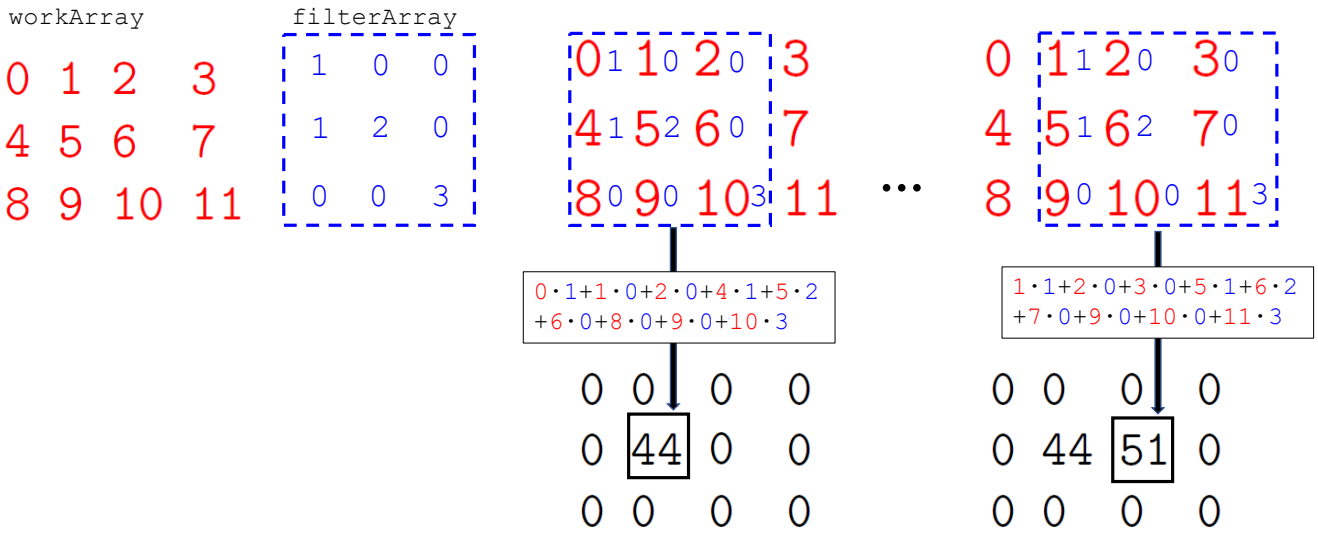


Abbildung 1: Veranschaulichung der einzelnen Schritte der Filteroperation für ein gegebenes `workArray` und `filterArray`. Um das Ergebnis für eine bestimmte Stelle zu berechnen, wird das `filterArray` mittig über das `workArray` gelegt. Das Ergebnis wird zuerst für die Stelle `[1][1]` berechnet, da 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 beliebten *Where's Waldo?* Kinderbücher[2].

Implementieren Sie folgende Aufgabenstellung:

- Implementieren Sie eine Methode `detectWaldo`:

  ```
  int[] detectWaldo(int[][] imgArrayGrayscale, int[][] templateArray)
  ```

  Die Methode übernimmt ein Grauwertbild `imgArrayGrayscale` und ein Template `templateArray` (Waldo), sucht das Template im Grauwertbild und gibt dessen Position als Bounding Box im Format `{x1,y1,x2,y2}` zurück. Die Koordinate (`x1,y1`) beschreibt dabei die linke, obere Ecke der Bounding Box, die Koordinate (`x2,y2`) die rechte, untere Ecke der Bounding Box (siehe Abbildung 3b). Analog zur Aufgabe 2 wandert das `templateArray` über das Bild `imgArrayGrayscale`, 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 aber nicht die Kreuz-Korrelation, sondern die *Summe der absoluten Differenzen* [3] (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).

  Analog zu Aufgabe 2 werden nur Positionen ausgewertet, bei denen das Template nicht über den Rand des Bildes hinausgeht. Wurde die Stelle mit minimaler Unähnlichkeit gefunden, wird die entsprechende Bounding Box zurückgegeben. Der von der Bounding Box beschriebene Bildauschnitt soll dabei die selbe Größe wie das `templateArray` haben.

  Sie können ihre Methode zunächst mit dem Eingabebild `waldo1.png` (Abbildung 2a) und dem Template `template1.png` (Abbildung 2b) testen (Hinweis: Die Bilddateien sind als Links hinterlegt, damit diese nicht bei jedem IntelliJ-Projekt in TUWEL hochgeladen werden müssen), wobei in diesem Fall eine Bounding Box mit den Werten `{567,98,601,112}` zurückgegeben werden sollte.

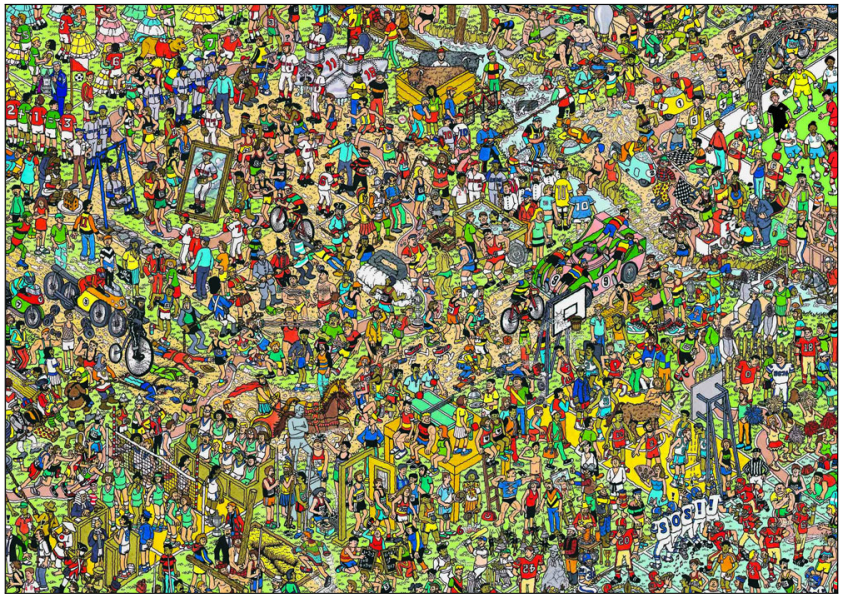
  Vorbedingungen: `imgArrayGrayscale != null`, `imgArrayGrayscale.length > 0`, für alle gültigen `i` gilt, dass `imgArrayGrayscale[i].length` dem selben konstanten Wert größer 0 entspricht; `templateArray != null`, `templateArray.length > 0` und ungerade, für alle gültigen `i` gilt, dass `templateArray[i].length` dem selben konstanten und ungeraden Wert größer 0 entspricht.

- In einem zweiten Schritt soll nun Waldo im Bild hervorgehoben werden, indem der Rest des Bildes, der sich außerhalb der Bounding Box befindet, verdunkelt wird. Verändern Sie dazu den in der `main` Methode markierten Code-Abschnitt. Befindet sich ein Pixel außerhalb

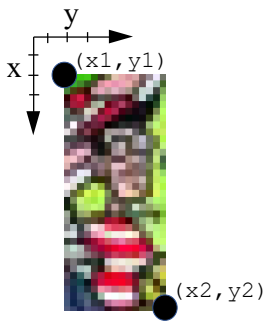
[2] `https://en.wikipedia.org/wiki/Where%27s_Wally%3F`

[3] `https://de.wikipedia.org/wiki/Summe_der_absoluten_Differenzen`

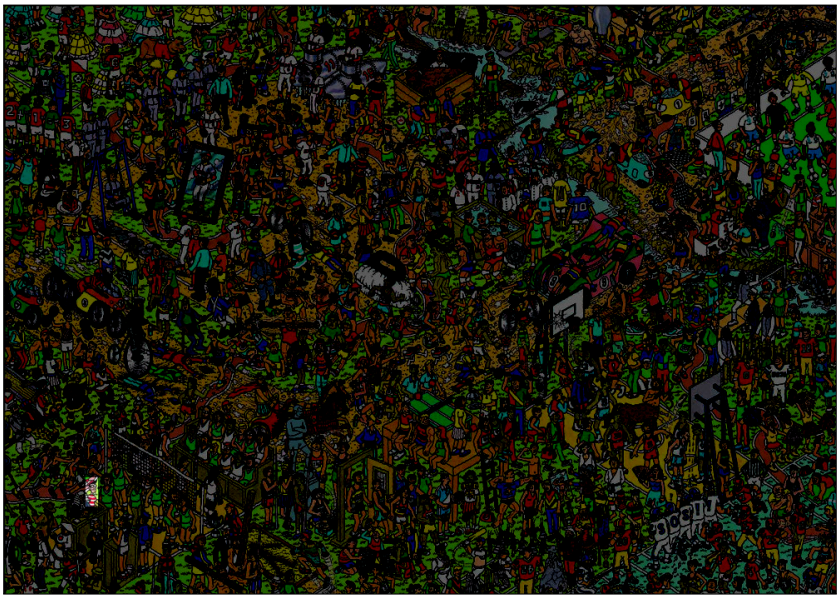
der Bounding Box, dann wird dessen Wert in allen drei Farbkanälen `imgArrayR`, `imgArrayG` und `imgArrayB` um den Wert 150 verringert. Wenn dabei ein negativer Wert entsteht, wird dieser Wert auf 0 gesetzt. Für das Bild `waldo1.png` und `template1.png` muss bei korrekter Implementierung das Ergebnis in Abbildung 2c herauskommen.



(a)



(b)



(c)

Abbildung 2: a) Eingabebild `waldo1.png`, b) Templatebild `template1.png` mit eingezeichnetem Koordinatensystem der Bounding Box und c) korrekte Ausgabe mit markiertem Waldo.

- 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.

Aufgabe 4 (2 Punkte)

Erweitern Sie die Aufgabe um folgende Funktionalität:

- Implementieren Sie eine Methode `genLandscape`:

  ```
  Color[][] genLandscape(int size)
  ```

  Diese Methode erzeugt eine Landschaft in einem zweidimensionalen Array vom Typ `Color` mit der Größe `size`×`size` und retourniert dieses Array. Dazu soll jedem Element im Array zufallsgeneriert entweder `Color.GRAY` (Felsen) oder `Color.GREEN` (Wiese) zugewiesen werden. Felsen sollen mit einer Wahrscheinlichkeit von 10% vorkommen (Bestimmung mittels `Math.random()`). Es entsteht dann eine Landschaft, wie in Abbildung 3a gezeigt.

- Implementieren Sie eine Methode `drawLandscape`:

  ```
  void drawLandscape(Color[][] landscape)
  ```

  Diese Methode zeichnet die Landschaft in einem Ausgabefenster von `canvasSize`×`canvasSize` Pixel. Jeder Eintrag des Arrays wird als gefülltes Quadrat gezeichnet.

- Implementieren Sie eine **rekursive** Methode `simLiquidFlow`:

  ```
  void simLiquidFlow(Color[][] landscape, int x, int y)
  ```

  Diese Methode generiert bzw. simuliert eine durch die Landschaft fließende Flüssigkeit. Die Flüssigkeit soll in der Landschaft `landscape` von oben nach unten mit der Breite eines Array-Elements fließen und die Landschaft an der Position `(x,y)` orange einfärben (`Color.ORANGE`). Die Flüssigkeit fließt generell mit einer Chance von 50% entweder nach links oder rechts unten. Immer wenn die Flüssigkeit auf einen grauen Felsen trifft (`Color.GRAY`), spaltet sich die Flüssigkeit links und rechts auf. Das heißt, dass die Flüssigkeit beim Felsen links und rechts vorbeifließt und danach zwei Flüsse weiterfließen. Zusätzlich wird das Pixel über dem Felsen immer orange eingefärbt. Der Felsen selbst wird mit `Color.BLACK` dunkler eingefärbt. Nach einer Spaltung fließen dann beide Flüssigkeitsstränge wieder mit einer Chance von 50% entweder nach links oder rechts unten. Das Ergebnis nach einem Aufruf von `simLiquidFlow` ist in Abbildung 3b dargestellt.

  Beachten Sie, dass durch die rekursiven Aufrufe auch folgende Situationen entstehen können: Trifft die Flüssigkeit auf einen schwarzen Felsen, dann passiert nichts, d.h. der Aufruf wird beendet. Trifft die Flüssigkeit auf einen orangen Array-Eintrag, dann bleibt dieser orange und der Aufruf wird weiter fortgesetzt, d.h. zwei Flüssigkeitsströme können sich an einer Stelle überschneiden und dann auch wieder trennen.

- Implementieren Sie eine **rekursive** Methode `simSpreadingFire`:

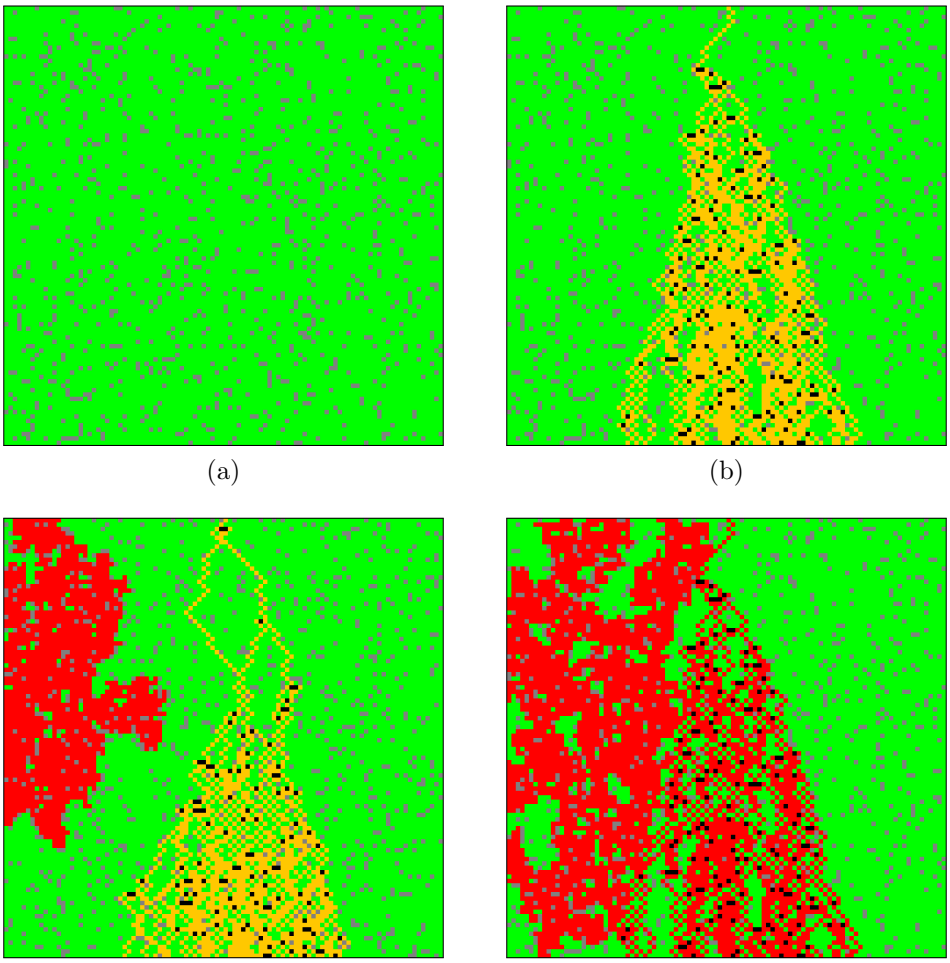
  ```
  void simSpreadingFire(Color[][] landscape, int x, int y)
  ```

Diese Methode simuliert ein Feuer und dessen Ausbreitung in einer gegebenen Landschaft `landscape`. Dazu wird ein Array-Eintrag an der Position `(x,y)`, der eine Wiese darstellt, entzündet. Danach soll sich das Feuer per Zufall in alle vier Himmelsrichtungen (Norden, Osten, Süden, Westen) ausbreiten und rot (`Color.RED`) eingefärbt werden. Für jede dieser 4 Richtungen wird sich das Feuer mit 60% Wahrscheinlichkeit ausbreiten, sofern es sich um eine Wiese handelt (Felsen (schwarz und grau) werden nicht entzündet und stoppen das Feuer). Trifft das Feuer auf die zuvor bereits in die Landschaft eingetragene orange Flüssigkeit, wird diese entzündet und färbt alle orangen Array-Einträge ebenfalls rot ein. Dazu soll die Methode `spreadFireInLiquid` (nachfolgend beschrieben) aufgerufen werden. Abbildung 3c zeigt ein Ergebnis, wo das Feuer die brennbare Flüssigkeit nicht erreicht hat. Abbildung 3d hingegen zeigt ein Endergebnis, wo das Feuer auch die Flüssigkeit erreicht und entzündet hat.

- Implementieren Sie eine **rekursive** Methode `spreadFireInLiquid`:

```
void spreadFireInLiquid(Color[][] landscape, int x, int y)
```

Diese Methode dient als Hilfsmethode, um beim Entzünden der Flüssigkeit alle orangen Array-Einträge in der Landschaft `landscape` rot einzufärben. Dazu muss, ausgehend von einem Eintrag `(x,y)`, in alle 8 Richtungen mit rekursiven Selbstaufrufen der Methode `spreadFireInLiquid(...)` gesucht werden, ob es noch einen weiteren orangen Eintrag gibt, der rot eingefärbt werden muss. Diese Methode entzündet nur die Flüssigkeit und nicht die neu angrenzenden Grasteile der Flüssigkeit. Achten Sie bei der Implementierung darauf, dass diese Methode keinerlei Schleifen enthält.



(a) (b)

(c) (d)

Abbildung 3: Landschaftsdarstellung nach a) Generierung, b) simulierter Flüssigkeit, c) simulierter Flüssigkeit mit Feuer und d) simulierter Flüssigkeit, die ebenfalls durch das Feuer entzündet wurde.