Aufgabenblatt 5

Kompetenzstufe 1 & Kompetenzstufe 2

Allgemeine Informationen zum Aufgabenblatt:

- Die Abgabe erfolgt in TUWEL. Bitte laden Sie Ihr IntelliJ-Projekt bis spätestens **Donnerstag, 06.01.2022 20:00 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 `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:

- 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 2 begonnen und in jeder weiteren Gegendiagonalen die Zahl um 2 vergrößert. Nach der Zahl `n` wird mit jeder weiteren Gegendiagonalen die Zahl wieder um 2 verringert, sodass es rechts unten wieder mit der Zahl 2 endet. Anschließend wird das neu erzeugte Array zurückgegeben.

  Vorbedingung: `n > 0`.

  Beispiele:

  `genFilledArray(2)` erzeugt →

```
2 4
4 2
```

  `genFilledArray(4)` erzeugt →

```
2 4 6 8
4 6 8 6
6 8 6 4
8 6 4 2
```

  `genFilledArray(5)` erzeugt →

```
2 4 6 8 10
4 6 8 10 8
6 8 10 8 6
8 10 8 6 4
10 8 6 4 2
```

- Implementieren Sie eine Methode `shiftLinesInArray`:

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

  Diese Methode baut ein ganzzahliges zweidimensionales Array `workArray` so um, dass alle Zeilen innerhalb des Arrays um eine Zeile nach oben verschoben werden. Die oberste Zeile wird ganz unten 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}})`
	`6 2 1`
	`0 7 9`
	`1 3 5`

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

- 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 immer vorne mit Einsen aufgefüllt, sodass alle Zeilen des neuen Arrays gleich viele Einträge haben.

  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}})`
	`1 1 1 4`
	`1 1 2 3`
	`1 1 5 6`
	`7 8 9 1`

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

- Implementieren Sie eine Methode `reformatArray`:

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

  Diese Methode interpretiert jede Zeile von `inputArray` als Dezimalzahl und erstellt ein neues eindimensionales Array vom Typ `long`, das die für jede Zeile entsprechende 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 10^0, das vorletzte Element jeder Zeile die Wertigkeit 10^1, 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 < 19` ist. Weiters gilt für alle gültigen `i` und `j`, dass `inputArray[i][j] >= 0` $\wedge$ `inputArray[i][j] <= 9` ist.

  Beispiele:

  `reformatArray(new int[][]{`
  `{3,5,6},`
  `{0,2,0}})` erzeugt $\rightarrow$

  Zeile 1: `{3,5,6}` $\rightarrow 10^2 * 3 + 10^1 * 5 + 10^0 * 6 = 356$
  Zeile 2: `{0,2,0}` $\rightarrow 10^2 * 0 + 10^1 * 2 + 10^0 * 0 = 20$

  ```
  356 20
  ```

  ```
  reformatArray(new int[][]{
  {1},
  {0, 2, 0},
  {1, 0, 2, 5, 0, 0, 0},
  {9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1}})
  ```
  erzeugt $\rightarrow$

  ```
  1 20 1025000 987654321010000000 1
  ```

Aufgabe 2 (1 Punkt)

Implementieren Sie folgende Aufgabenstellung:

- Implementieren Sie eine Methode `genMeanFilter`:

  ```
  double[][] genMeanFilter(int rows, int cols)
  ```

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

  Beispiele:

  `genMeanFilter(3, 3)` erzeugt →

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

  `genMeanFilter(1, 5)` erzeugt →

  ```
  0,20 0,20 0,20 0,20 0,20
  ```

- 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 allen Positionen über `workArray` gelegt wird. 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. Punkte außerhalb des Rands von `filterArray` und `workArray` werden nicht berücksichtigt.

  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 Positionen von `workArray` durchgeführt. Das Ergebnis-Array würde in diesem Fall folgendermaßen aussehen:

```
15 20 26 8
35 44 51 22
16 30 34 38
```

Beachten Sie, dass an den Randstellen von `workArray` der Fall auftritt, dass es nicht für alle Positionen im `filterArray` ein korrespondierendes Element im `workArray` gibt. So können beispielsweise an der Stelle `[0][0]` nur 4 Produkte berechnet werden, die restlichen 5 werden ignoriert.

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:

  ```
  1 0 0
  ```

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

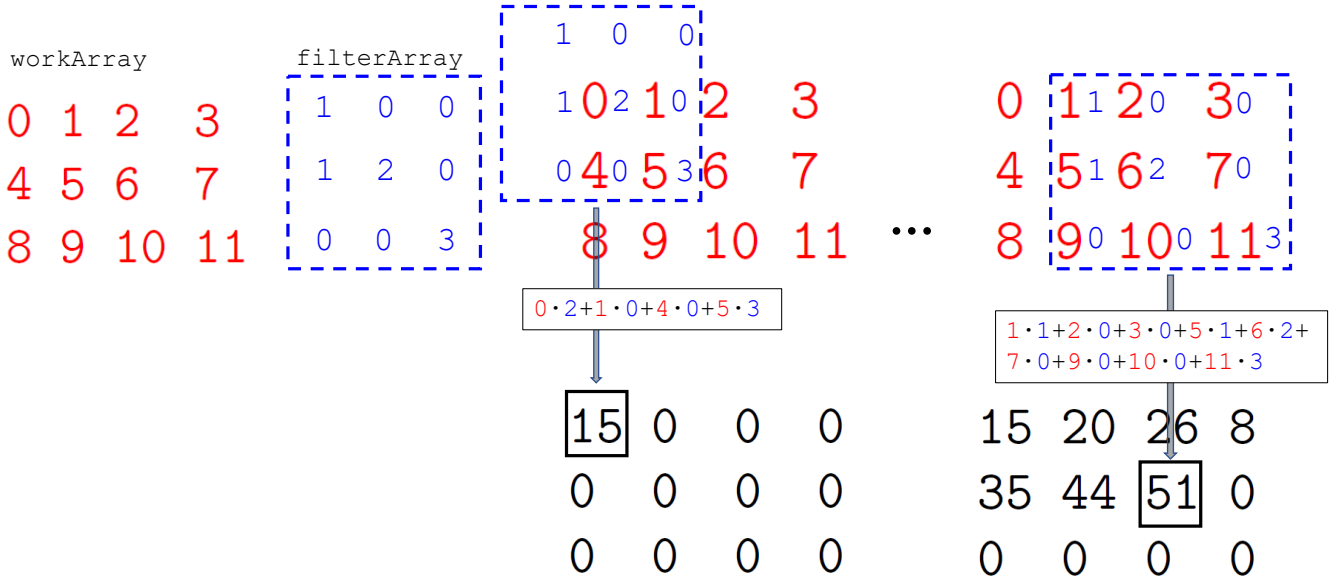


Abbildung 1: Veranschaulichung der einzelnen Schritte der Filteroperation für ein gegebenes `workArray` und `filterArray`. Um das Ergebnis für eine bestimmte Stelle zu berechnen, wird das `filterArray` mittig über das `workArray` gelegt. Das Ergebnis wird zuerst für die Stelle `[0][0]` berechnet. Hier werden nun die korrespondierenden Elemente im `workArray` und `filterArray` multipliziert und aufsummiert (Ergebnis: 15; die Elemente, die über den Rand hinausgehen, werden ignoriert). Diese Operation wird nun für alle Positionen von `workArray` wiederholt. An der Stelle `[1][2]` lautet das Ergebnis beispielsweise 51.

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 Regionen zu finden, die ähnlich zu einer definierten Template-Region sind.

Implementieren Sie eine Methode `blackenSimilarRegions`:

```
int[][] blackenSimilarRegions(int[][] imgArray, int rowStart, int rowEnd,
                              int colStart, int colEnd, double threshold)
```

Vorbedingungen: `imgArray != null`, `imgArray.length > 0`, für alle gültigen `i` gilt, dass `imgArray[i].length` dem selben konstanten Wert größer 0 entspricht; `rowStart` und `rowEnd` sind `>= 0` und `< imgArray.length`; `rowStart <= rowEnd`; `colStart` und `colEnd` sind `>= 0` und `< imgArray[0].length`; `colStart <= colEnd`.

Die Methode übernimmt ein Grauwertbild `imgArray` und sucht Bildbereiche, die dem durch die Koordinaten `rowStart`, `rowEnd`, `colStart` und `colEnd` definierten Bildbereich ähnlich sind. Der rechteckige Bildbereich beinhaltet die Zeilen von `rowStart` bis inklusive `rowEnd` und die Spalten von `colStart` bis inklusive `colEnd` des Arrays `imgArray`. Dieser Bildbereich mit `(rowEnd - rowStart + 1)` Zeilen und `(colEnd - colStart + 1)` Spalten wird aus dem Bild extrahiert und dient nun als Template, um ähnliche Bildbereiche im Bild zu suchen und zu schwärzen. Analog zu Aufgabe 2 wandert das Template über das Bild `imgArray`, 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 Filter-Operation, sondern die *Abweichungsquadratsumme* [2] (*Sum of Squared Deviations* - SSD): alle korrespondierenden Pixelwerte werden subtrahiert und die Quadrate all dieser Differenzen werden aufsummiert. Diese Vorgehensweise liefert uns für jeden Bildpunkt ein Maß der *Unähnlichkeit*, und wir können das Template überall dort finden, wo dieses Maß unter dem Schwellwert `threshold` liegt.
Es reicht in diesem Fall, nur Positionen auszuwerten, bei denen das Template nicht über den Rand des Bildes hinausgeht. Wurde eine Stelle mit einer Unähnlichkeit kleiner dem Schwellwert gefunden, wird der entsprechende Bildbereich geschwärzt (d.h., alle Werte werden auf 0 gesetzt). Nachdem alle Bildbereiche überprüft wurden, wird das geschwärzte Bild zurückgeliefert.

Achten Sie darauf, dass Ihre Implementierung nicht das übergebene Originalbild `imgArray` ändert!

Zum Testen Ihrer Implementierung gibt es in `main` drei vordefinierte Aufrufe von `blackenSimilarRegions`. Das Testbild entspricht dabei einer Seite von Aufgabenblatt 1. Der erste Aufruf sollte bei korrekter Implementierung alle Zeichen "g" schwärzen, der zweite Aufruf alle Wörter "while". Beim dritten Aufruf besteht des Template aus nur einem Pixel und bewirkt eine Art Binarisierung des Bildes (alle grauen und schwarzen Pixel werden geschwärzt). Abbildung 2 zeigt das Eingabebild, das "g"-Template des Aufrufs `blackenSimilarRegions(imgArray, 148, 158, 321, 328, 1e5)` und das Ergebnis der Schwärzung.

[2] `https://de.wikipedia.org/wiki/Summe_der_Abweichungsquadrate`

Einführung in die Programmierung 1 185.A91 - WS2021

Aufgabe 3 (1 Punkt)

Erweitern Sie die Methode `main`:

- Deklarieren Sie eine String-Variable `text` und initialisieren Sie diese mit `"Die Sonne scheint in vielen Ländern."`. Testen Sie zusätzlich mit dem String `"Sommer, Sonne, Kaktus"`, um Ihre Implementierung zu prüfen.

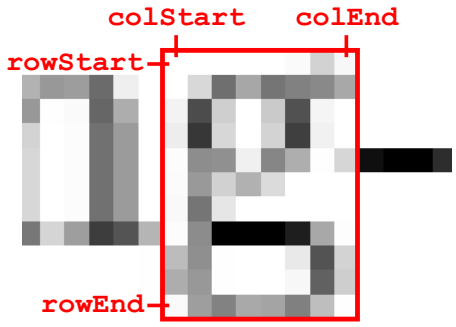
a) Schreiben Sie eine while-Schleife, die den String `text` von vorne beginnend durchläuft und Zeichen für Zeichen nebeneinander ausgibt, bis der Buchstabe `'n'` 3-Mal gefunden wurde. Gab es weniger als 3 Vorkommen von `'n'`, dann wird der komplette String ausgegeben.
Erwartetes Ergebnis:
`"Die Sonne scheint in vielen Ländern."` **liefert** `"Die Sonne schein"`
`"Sommer, Sonne, Kaktus"` **liefert** `"Sommer, Sonne, Kaktus"`

b) Schreiben Sie eine while-Schleife, die im String `text` von vorne beginnend das erste Vorkommen des Buchstabens `'l'` sucht. Wird der Buchstabe `'l'` gefunden, dann wird dessen Index auf der Konsole ausgegeben. Andernfalls wird -1 ausgegeben. Für diese Implementierung dürfen die Methoden `indexOf(...)` und `lastIndexOf(...)` **nicht** verwendet werden.
Erwartetes Ergebnis:
`"Die Sonne scheint in vielen Ländern."` **liefert** 24
`"Sommer, Sonne, Kaktus"` **liefert** -1

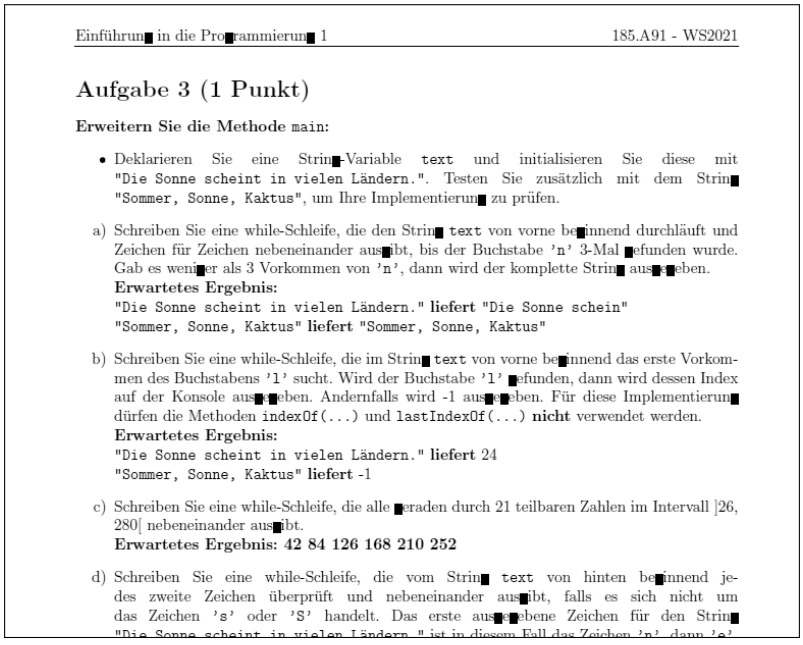
c) Schreiben Sie eine while-Schleife, die alle geraden durch 21 teilbaren Zahlen im Intervall]26, 280[nebeneinander ausgibt.
Erwartetes Ergebnis: 42 84 126 168 210 252

d) Schreiben Sie eine while-Schleife, die vom String `text` von hinten beginnend jedes zweite Zeichen überprüft und nebeneinander ausgibt, falls es sich nicht um das Zeichen `'s'` oder `'S'` handelt. Das erste ausgegebene Zeichen für den String `"Die Sonne scheint in vielen Ländern."` ist in diesem Fall das Zeichen `'n'`, dann `'e'`

(a)



(b)



Einführung in die Programmierung 1 185.A91 - WS2021

Aufgabe 3 (1 Punkt)

Erweitern Sie die Methode `main`:

- Deklarieren Sie eine String-Variable `text` und initialisieren Sie diese mit `"Die Sonne scheint in vielen Ländern."`. Testen Sie zusätzlich mit dem String `"Sommer, Sonne, Kaktus"`, um Ihre Implementierung zu prüfen.

a) Schreiben Sie eine while-Schleife, die den String `text` von vorne beginnend durchläuft und Zeichen für Zeichen nebeneinander ausgibt, bis der Buchstabe `'n'` 3-Mal gefunden wurde. Gab es weniger als 3 Vorkommen von `'n'`, dann wird der komplette String ausgegeben.
Erwartetes Ergebnis:
`"Die Sonne scheint in vielen Ländern."` **liefert** `"Die Sonne schein"`
`"Sommer, Sonne, Kaktus"` **liefert** `"Sommer, Sonne, Kaktus"`

b) Schreiben Sie eine while-Schleife, die im String `text` von vorne beginnend das erste Vorkommen des Buchstabens `'l'` sucht. Wird der Buchstabe `'l'` gefunden, dann wird dessen Index auf der Konsole ausgegeben. Andernfalls wird -1 ausgegeben. Für diese Implementierung dürfen die Methoden `indexOf(...)` und `lastIndexOf(...)` **nicht** verwendet werden.
Erwartetes Ergebnis:
`"Die Sonne scheint in vielen Ländern."` **liefert** 24
`"Sommer, Sonne, Kaktus"` **liefert** -1

c) Schreiben Sie eine while-Schleife, die alle geraden durch 21 teilbaren Zahlen im Intervall]26, 280[nebeneinander ausgibt.
Erwartetes Ergebnis: 42 84 126 168 210 252

d) Schreiben Sie eine while-Schleife, die vom String `text` von hinten beginnend jedes zweite Zeichen überprüft und nebeneinander ausgibt, falls es sich nicht um das Zeichen `'s'` oder `'S'` handelt. Das erste ausgegebene Zeichen für den String `"Die Sonne scheint in vielen Ländern."` ist in diesem Fall das Zeichen `'n'`, dann `'e'`

(c)

Abbildung 2: a) Teil des Eingabebildes, b) extrahiertes “g”-Template und c) Ergebnis mit geschwärzten Buchstaben “g” (beachten Sie, dass hier nur Buchstaben der selben Größe und mit dem selben Schriftgrad geschwärzt werden).

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 20% vorkommen. Verwenden Sie dazu die Klasse `Random`[3]. Das Objekt `myRand` muss dazu verwendet werden, um mit `myRand.nextDouble()` eine Zufallszahl zwischen 0.0 (inklusive) und 1.0 (exklusive) zu erzeugen. Wenn zum Beispiel die Abfrage `myRand.nextDouble() < 0.2 true` ergibt, dann wird ein Felsen (`Color.GRAY`) in das Array eingetragen. Es entsteht bei Verwendung von `myRand.nextDouble() < 0.2` eine Landschaft, wie in Abbildung 3a gezeigt.

  Vorbedingung: `size > 0`.

- Implementieren Sie eine Methode `drawLandscape`:

  ```
  void drawLandscape(CodeDraw myDrawObj, Color[][] landscape)
  ```

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

  Vorbedingungen: `landscape != null`, `landscape.length > 0` und `landscape.length` ist gleich `landscape[i].length` für alle gültigen `i`.

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

  ```
  void simWaterFlow(Color[][] landscape, int row, int col)
  ```

  Diese Methode generiert bzw. simuliert einen durch die Landschaft fließenden Fluss. Der Fluss soll in der Landschaft `landscape` von oben nach unten mit der Breite eines Array-Elements fließen und die Landschaft an der Position `(row,col)` blau einfärben (`Color.BLUE`). Das Wasser fließt generell mit einer Chance von 50% entweder nach rechts oder links unten. Wenn die Abfrage `myRand.nextDouble() < 0.5 true` liefert, dann soll der Fluss nach rechts unten weiterfließen, ansonsten nach links unten.

  Immer wenn die Flüssigkeit auf einen grauen Felsen trifft (`Color.GRAY`), spaltet sich die Flüssigkeit links und rechts auf. Das heißt, dass direkt links und rechts neben dem Felsen (in der selben Zeile) zwei neue Wasserstränge entstehen und der bestehende Wasserstrang gestoppt wird. Nach einer Spaltung fließen dann beide Wasserstränge für sich wieder mit einer Chance von 50% nach links oder rechts unten weiter. Der Felsen selbst wird mit `Color.DARK_GRAY` eingefärbt. Das Ergebnis nach einem Aufruf von `simWaterFlow` ist in Abbildung 3b dargestellt.

[3] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html`

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

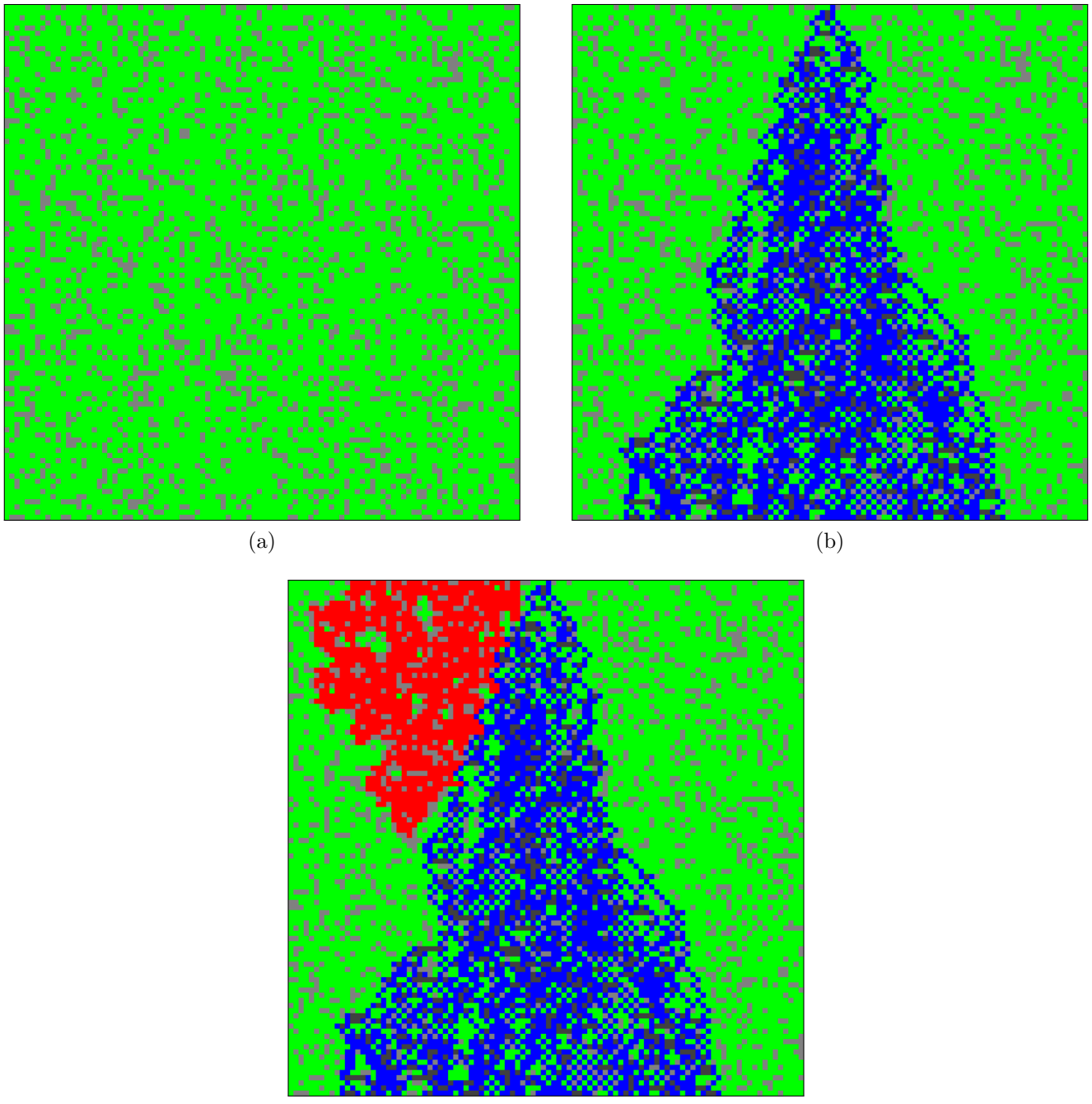
Vorbedingungen: `landscape != null`, `landscape.length > 0` und `landscape.length` ist gleich `landscape[i].length` für alle gültigen `i`. `row >= 0` und `row < landscape.length`. `col >= 0` und `col < landscape[i].length`.

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

```
void simSpreadingFire(Color[][] landscape, int row, int col)
```

Diese Methode simuliert ein Feuer und dessen Ausbreitung in einer gegebenen Landschaft `landscape`. Dazu wird ein Array-Eintrag an der Position `(row,col)`, der eine Wiese darstellt, entzündet. Danach soll sich das Feuer per Zufall in alle vier Himmelsrichtungen (Reihenfolge der Ausbreitung: Norden, Osten, Süden, Westen) ausbreiten und die Landschaft rot (`Color.RED`) eingefärbt werden. Für jede einzelne dieser 4 Richtungen getrennt wird sich das Feuer mit 70% Wahrscheinlichkeit (verwenden Sie `myRand.nextDouble() >= 0.3` für die Ausbreitung in jede der 4 Himmelsrichtungen) ausbreiten, sofern es sich um eine Wiese handelt. Die Felsen (grau und dunkelgrau) werden nicht entzündet und stoppen das Feuer. Trifft das Feuer auf das zuvor bereits in die Landschaft eingetragene blaue Wasser, dann wird das Feuer ebenfalls gestoppt. Abbildung 3c zeigt ein Ergebnis, bei dem das Feuer sich ausgebreitet hat.

Vorbedingungen: `landscape != null`, `landscape.length > 0` und `landscape.length` ist gleich `landscape[i].length` für alle gültigen `i`. `row >= 0` und `row < landscape.length`. `col >= 0` und `col < landscape[i].length`.



(a) (b)

(c)

Abbildung 3: Landschaftsdarstellung nach a) Generierung der Felsen, b) simuliertes fließendes Wasser und c) simuliertes fließendes Wasser mit Feuer.