Aufgabenblatt 5

Kompetenzstufe 1 & Kompetenzstufe 2

Allgemeine Informationen zum Aufgabenblatt:

- Die Abgabe erfolgt in TUWEL. Bitte laden Sie Ihr IntelliJ-Projekt bis spätestens **Mittwoch, 10.12.2025 23:55 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
- Grafische Ausgabe
- Zweidimensionale Arrays und Bilder

Aufgabe 1 (1 Punkt)

Implementieren Sie folgende Aufgabenstellung:

- Implementieren Sie eine Methode `changeRows`:

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

  Diese Methode baut ein ganzzahliges, zweidimensionales Array `workArray` mit genau drei Zeilen so um, dass die kürzeste Zeile an die erste Stelle (Index 0) und die längste Zeile an die letzte Stelle (Index 2) des Arrays verschoben wird. Das heißt, dass die Zeilen umgehängt (Referenz wird kopiert) werden und *nicht* neue Zeilen entstehen dürfen. Die mittlere Zeile wird neu erstellt und enthält alle Elemente der kürzesten Zeile, gefolgt von allen Elementen der längsten Zeile. Die neue Zeile wird dem Index 1 in `workArray` zugewiesen. Sind zwei Zeilen gleich lang, dann wird jene mit kleinerem Index für den Tausch verwendet. Sollten alle drei Zeilen gleich lang sein, dann wird keine Veränderung am Array vorgenommen. Sie dürfen aber innerhalb der Methode temporäre Array-Variablen anlegen.

  Vorbedingungen: `workArray != null`, `workArray.length == 3` und für alle gültigen `i` gilt `workArray[i] != null` und `workArray[i].length > 0`.

  Beispiele:

Aufruf	Ergebnis	Aufruf	Ergebnis

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

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

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

Aufgabe 2 (1 Punkt)

Implementieren Sie folgende Aufgabenstellung:

- Implementieren Sie eine Methode `reformatArray`:

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

  Die Methode verändert bis auf die erste Zeile alle Zeilen von `workArray`. Dazu wird in jeder Zeile das größte Element gesucht. In der Folgezeile wird das bis dahin größte gefundene Element angehängt. Das heißt, dass ab der zweiten Zeile jede Zeile mit der größten Zahl, die bis zur vorhergehenden Zeile gefunden wurde, erweitert wird. Hat `workArray` nur eine Zeile wird keine Veränderung am Array vorgenommen.

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

  Beispiele:

Aufruf	Ergebnis

`reformatArray(new int[][]{`
`{5, 4, 6, 9},`
`{1, 10, 5, 7},`
`{3},`
`{2, 8},`
`{11, 8, 10},`
`{9}})`
	`5 4 6 9`
	`1 10 5 7 9`
	`3 10`
	`2 8 10`
	`11 8 10 10`
	`9 11`

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

`reformatArray(new int[][]{`
`{6, 3, 4, 1},`
`{8, 1, 2},`
`{9, 10},`
`{14},`
`{13, 10}})`
	`6 3 4 1`
	`8 1 2 6`
	`9 10 8`
	`14 10`
	`13 10 14`

`reformatArray(new int[][]{` `{5, 3, 4, 1, 8}})`	`5 3 4 1 8`

Aufgabe 3 (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 `0,00` und `1,00` enthält. Jedes Element, dessen Abstand zum Mittelpunkt des Arrays kleiner als `radius` ist, wird auf `1,00` gesetzt, die übrigen Elemente auf `0,00`. 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 $\sqrt{(0-1)^2 + (0-1)^2}$ größer als 1.2 ist. Der Wert an der Stelle `[0][1]` wird auf 1 gesetzt, da sein Abstand zum Mittelpunkt $\sqrt{(1-1)^2 + (0-1)^2}$ 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. Zusätzlich zeigt Abbildung 2 ein größeres Beispiel mit einem 7×7 `workArray` und die Anwendung eines 3×3 Filters (`filterArray`).

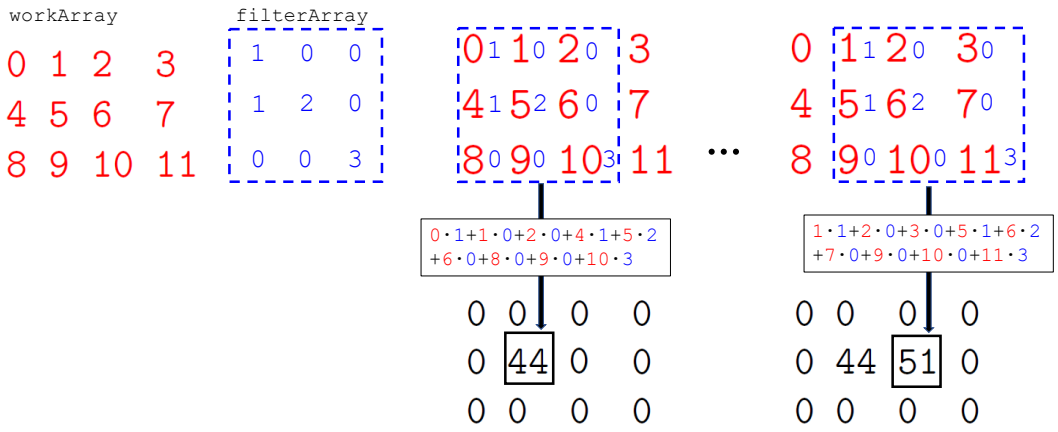


Abbildung 1: Veranschaulichung der einzelnen Schritte der Filteroperation für ein gegebenes `workArray` und `filterArray`. Um das Ergebnis für eine bestimmte Stelle zu berechnen, wird das `filterArray` mittig über die entsprechende Stelle im `workArray` gelegt. Das Ergebnis wird zuerst für die Stelle `[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).

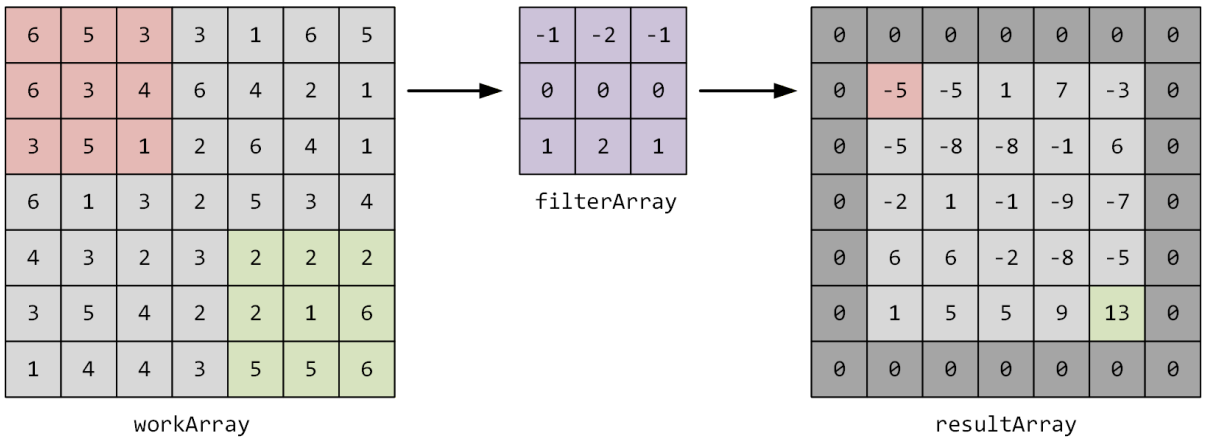


Abbildung 2: Beispiel für die Anwendung eines 3×3 Filters auf ein 7×7 Array. Auf der linken Seite ist das Ausgangsarray `workArray`, in der Mitte das `filterArray` und rechts das Ergebnis in einem `resultArray`. Durch die Filtergröße gibt es immer einen Rand, der unbearbeitet bleibt (siehe rechts dunkelgrauer Bereich). Die Filteroperation beginnt links oben ([1][1]) und arbeitet danach Zeile für Zeile (Index für Index) weiter und endet rechts unten ([5][5]).

- 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.

Aufgabe 4 (2 Punkte)

Bei dieser Aufgabe soll ähnlich wie bei Aufgabe 3 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` demselben 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 3 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 - Sum of Squared Deviations (SSD)*: Hier wird für alle Pixelwerte (x, y) in `imgArray` das Template überlagert. Dann werden alle korrespondierenden Pixelwerte des Templates und des Bildes (`imgArray`) 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 (SSD-Wert) unter dem Schwellwert `threshold` liegt. In Abbildung 3 finden Sie ein Beispiel mit einem 3×3 Template. Man sieht rechts den grün eingefärbten SSD-Wert. Das ist der kleinste Wert und somit ist dort die größte Ähnlichkeit zwischen dem Bild (`imgArray`) und dem Template gegeben. Dieser Wert 3 ergibt sich aus der Berechnung $(1-2)^2+(3-3)^2+(2-1)^2+(3-4)^2+(2-2)^2+(3-3)^2+(5-5)^2+(4-4)^2+(2-2)^2 = 1+0+1+1+0+0+0+0+0 = 3$.
Es reicht in diesem Fall, nur Positionen auszuwerten, bei denen das Template nicht über den Rand des Bildes hinausgeht. Wenn ein Bereich mit einer Unähnlichkeit kleiner dem Schwellenwert gefunden wurde, wird der entsprechende Bereich geschwärzt und alle Werte auf 0 gesetzt. Das heißt, dass alle Werte im Bereich, bei dem der Filter zur Gänze in `imgArray` ist und bei dem der berechnete SSD-Wert kleiner als der `threshold` ist, schwarz eingefärbt werden. Nachdem alle Bildbereiche überprüft wurden, wird das geschwärzte Bild zurückgeliefert.

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

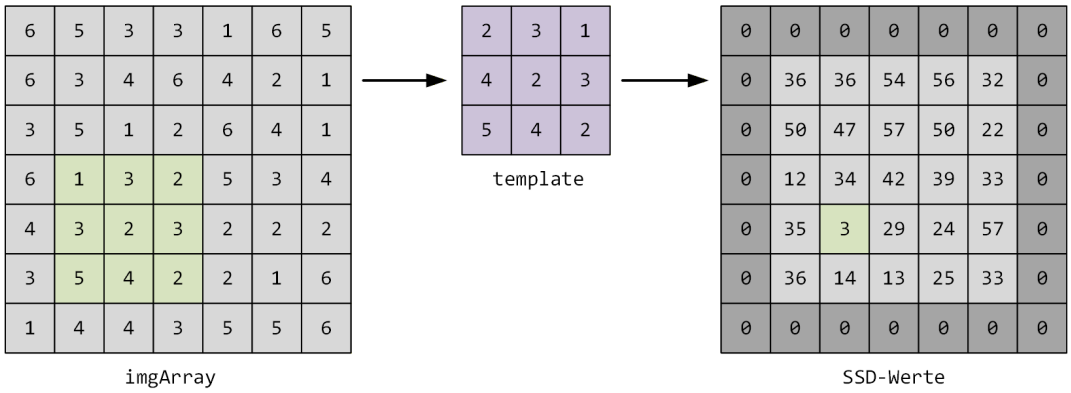


Abbildung 3: Beispiel für die Ähnlichkeitswerte eines 3×3 Templates mit einem 7×7 Bild. Auf der linken Seite ist das Originalbild `imgArray`, in der Mitte das Template `template` und rechts sind die SSD-Werte dargestellt.

Zum Testen Ihrer Implementierung gibt es in `main` drei vordefinierte Aufrufe von `blackenSimilarRegions`. Das Testbild entspricht dabei einer Seite von einem Aufgabenblatt in einem vorherigen Semester. Der erste Aufruf sollte bei korrekter Implementierung alle Zeichen "g" schwärzen, der zweite Aufruf alle Wörter "while". Beim dritten Aufruf besteht das Template aus nur einem Pixel und bewirkt eine Art Binarisierung des Bildes (alle grauen und schwarzen Pixel werden geschwärzt). Abbildung 4 zeigt das Eingabebild, das "g"-Template des Aufrufs `blackenSimilarRegions(imgArray, 148, 158, 321, 328, 1e5)` und das Ergebnis der Schwärzung.

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.

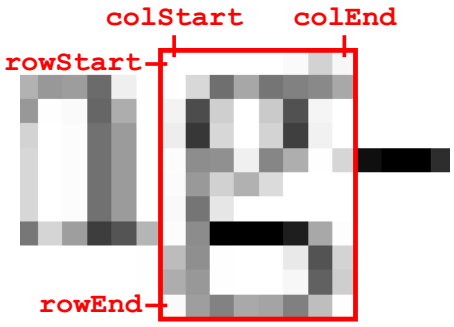
a) Schreiben Sie eine while-Schleife, die den String `text` von vorne beginnend durchläuft und Zeichen für Zeichen nebeneinander ausgibt, bis der Buchstabe `'n'` 3-Mal gefunden wurde. Gab es weniger als 3 Vorkommen von `'n'`, dann wird der komplette String ausgegeben.
Erwartetes Ergebnis:
`"Die Sonne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

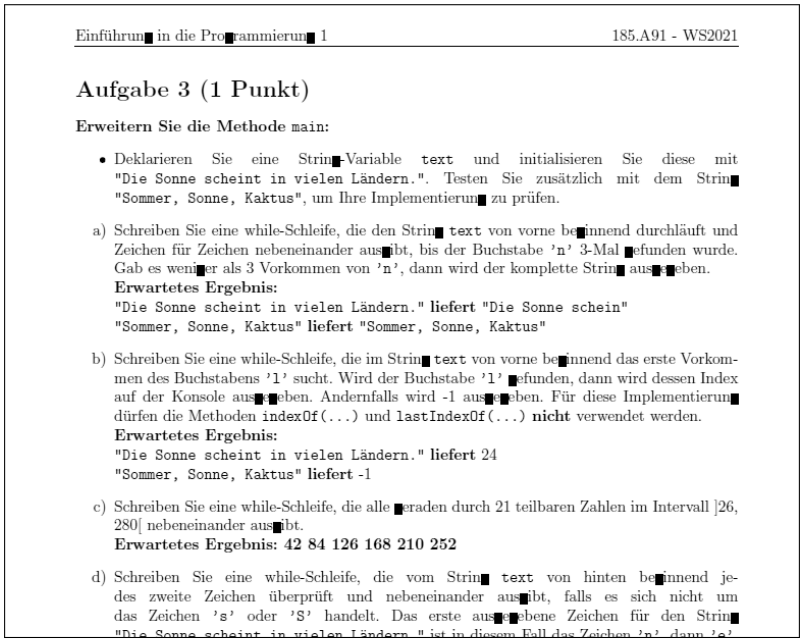
c) Schreiben Sie eine while-Schleife, die alle geraden durch 21 teilbaren Zahlen im Intervall]26, 280[nebeneinander ausgibt.
Erwartetes Ergebnis: 42 84 126 168 210 252

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 4: a) Teil des Eingabebildes, b) extrahiertes “g”-Template und c) Ergebnis mit geschwärzten Buchstaben “g” (beachten Sie, dass hier nur Buchstaben derselben Größe und mit demselben Schriftgrad geschwärzt werden).