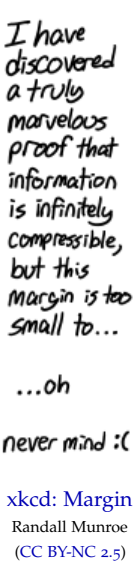
252-0027-00: Einführung in die Programmierung

Übungsblatt 7

Abgabe: 04. November 2025, 23:59

Checken Sie mit IntelliJ wie bisher die neue Übungsvorlage aus. Importieren Sie beide IntelliJ-Projekte (das Projekt für den Bonus und das Projekt für die restlichen Aufgaben). Beachten Sie, dass Sie mehrere unabhängige Programme im bonusunabhängigen IntelliJ-Projekt haben werden. Bevor Sie ein Programm starten, achten Sie deshalb darauf, dass Sie die richtige Datei ausgewählt oder im Editor geöffnet haben. *Vergessen Sie nicht, Ihren Programmcode zu kommentieren!*

Ab Woche 5, werden die TAs Ihnen nur dann Feedback geben, wenn Sie eine entsprechende Datei namens "requestfeedback.txt" in dem Projekt-Ordner hochladen. In der Text-Datei geben Sie an, für welche Aufgaben Sie Feedback erhalten möchten. Um eine neue Text-Datei in IntelliJ zu erstellen, rechts-klicken Sie den uXX Projekt-Ordner und wählen *New* → *File*. Im erscheinenden Fenster, benennen Sie die Datei mit "requestfeedback.txt" und drücken die Enter-Taste. Danach wählen Sie *Add*, falls IntelliJ Sie fragt, ob Sie die Datei zu git hinzufügen möchten.



xkcd: Margin
Randall Munroe
(CC BY-NC 2.5)

Aufgabe 1: Dreiecksmatrix

Die Klasse `Triangle` erlaubt die Darstellung von $Z \times S$ Dreiecksmatrizen (von `int` Werten). Z und S sind immer strikt grösser als 1 (d.h., > 1). Eine $Z \times S$ Dreiecksmatrize hat Z Zeilen $X_0, X_1, ..., X_{Z-1}$, wobei Zeile X_i genau $(i * (S-1)/(Z-1)) + 1$ viele Elemente hat. Dieser Ausdruck wird nach den Regeln für `int` Ausdrücke in Java ausgewertet. Für eine Dreiecksmatrix `d` ist $\texttt{d}_{i,j}$ das $(j+1)$-te Element in der $(i+1)$-ten Zeile. $\texttt{d}_{0,0}$ ist das erste Element in der ersten Zeile [die immer genau 1 Element hat]. Abbildung 1 zeigt Beispiele von Dreiecksmatrizen. Beachten Sie, dass es möglich ist, dass zwei (aufeinanderfolgende) Zeilen die selbe Anzahl Elemente haben.

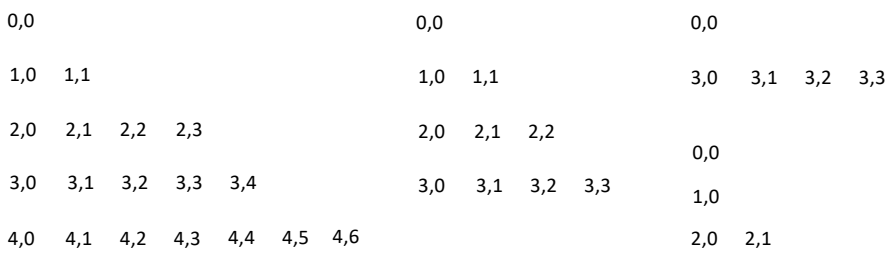


Abbildung 1: Beispiele von 5×7, 4×4, 2×4 und 3×2 Dreiecksmatrizen.

In der Datei `Triangle.java` finden Sie die Klasse `Triangle` mit einem Konstruktor `Triangle(int z, int s)`, der eine `z` $\times$ `s` Dreiecksmatrix erstellt. Dieser Konstruktor setzt die Werte aller Elemente auf 0. Vervollständigen Sie diese Klasse, so dass die folgenden Methoden unterstützt werden:

1. `int get(int i, int j)` gibt das Element $\texttt{d}_{\texttt{i,j}}$ zurück.
2. `void put(int i, int j, int value)` setzt das Element $\texttt{d}_{\texttt{i,j}}$ auf den Wert `value`.
3. `int[] linear()` liefert die Elemente in der kanonischen Reihenfolge (die Elemente jeder Zeile mit steigendem Index, und die Zeilen in steigender Reihenfolge).
4. `void init(int[] data)` ersetzt die Elemente von `d` durch die Werte in `data`. Sie dürfen annehmen, dass `data` genauso viele Elemente hat wie `d`. Die Methode setzt die Elemente von `D`, so dass die Folge `d.init(data); int[] y = d.linear();` in einen Array `y` resultiert für den `Arrays.equals(y, data)` den Wert `true` ergibt.
5. `void add(Triangle t)` Ein Aufruf `d.add(t)` addiert zu jedem Element $\texttt{d}_{i,j}$ den Wert von $\texttt{t}_{i,j}$, falls $\texttt{t}_{i,j}$ existiert. Falls $\texttt{t}_{i,j}$ nicht existiert, dann bleibt $\texttt{d}_{i,j}$ unverändert.

Tests finden Sie in der Datei "TriangleTest.java". Die Datei "TriangleGradingTest.java" enthält die Tests, welche wir bei der Prüfung für die Korrektur verwendet haben. Wir empfehlen, diese Tests erst zu verwenden, wenn Sie denken, dass Ihre Lösung korrekt ist, damit Sie sehen können, wie Sie bei einer Prüfung abgeschnitten hätten.

Aufgabe 2: Hoare Tripel

Welche dieser Hoare-Tripel sind (un)gültig? Bitte geben Sie für ungültige Tripel ein Gegenbeispiel an. Die Anweisungen sind Teil einer Java-Methode. Alle Variablen sind vom Typ `int` und es gibt keinen Overflow. Speichern Sie diese in der Text-Datei "HoareTriple.txt".

1. `{ x >= 0 || y >= 0 }  z = x * y;  { z > 0 }`
2. `{ x > 10 }  z = x % 10;  { z > 0 }`
3. `{ x > 0 }  y = x * x; z = y / 2;  { z > 0 }`

4. `{ x > 0 }  y = x * x; sum = y % (x + 1); { sum > 1 }`

5. `{ b > c }`

```
    if (x > b) {
        a = x;
    } else {
        a = b;
    }
```

`{ a > c }`

6. `{ x > 10 }`

```
      if (x % 4 == 0) {
          y = x / 4;
      }
```

`{ y >= 2 }`

7. `{ y > 0 }`

```
      if (x > y) {
          x = x - y;
          y = y * 2;
      } else {
          x = x * x;
          y = y - x;
      }
```

`{ y > 0 && x > 0 }`

8. `{ x != 0 }`

```
      if (x % 2 == 0) {
          x = x / 2;
      } else if (x % 3 == -1) {
          x = x * 3;
      } else {
          x = x + 1;
      }
```

`{ x != 0 }`

Aufgabe 3: Weakest Precondition

Bitte geben Sie für die folgenden Programmsegmente die schwächste Vorbedingung (weakest precondition) an. Bitte verwenden Sie Java-Syntax. Alle Anweisungen sind Teil einer Java Methode. Alle Variablen sind vom Typ `int` und es gibt keinen Overflow. Speichern Sie diese in der Text-Datei "WeakestPrecondtion.txt".

1.
```
P: { ?? }
S:   if (x < 5) {
       y = x * x;
     } else {
       y = x + 1;
     }
Q: { y >= 9 }
```

2.
```
P: { ?? }
S:  if (x != y) {
      x = y;
    } else {
      x = y + 1;
    }
Q: { x != y }
```

3.
```
P: { ?? }
S:  if (x > y) {
      min = y;
    } else {
      min = x;
    }
Q: { min <= x && min <= y }
```

4.
```
P: { ?? }
S:  if (x != 0) {
      z = x;
    } else {
      z = x + 1;
    }
Q: { z > 0 }
```

Aufgabe 4: Blackbox Testing

Im letzten Übungsblatt haben Sie Testautomatisierung mit JUnit kennengelernt. In dieser Aufgabe sollen Sie nun Tests für eine Methode schreiben, deren Implementierung Sie nicht kennen. Dadurch werden Sie weniger durch möglicherweise falsche Annahmen beeinflusst, die bei einer Implementierung getroffen wurden. Sie müssen sich also überlegen, wie sich *jede* fehlerfreie Implementierung verhalten muss. Diesen Ansatz nennt man auch Black-Box Testing da die Details der Implementation wie in einer blickdichten Box versteckt und daher nicht einsehbar sind.

In Ihrem "U06"-Projekt befindet sich eine "blackbox.jar"-Datei, welche eine kompilierte Klasse `BlackBox` enthält. Den Code dieser Klasse können Sie nicht sehen, aber sie enthält eine Methode `void rotateArray(int[] values, int steps)`, welche Sie aus einer eigenen Klasse oder einem Unit-Test aufrufen können. Diese Methode "rotiert" ein `int`-Array um eine gegebene Anzahl Schritte.

Vereinfacht macht die Methode `rotateArray()` Folgendes: Eine Rotation mit `steps=1` bedeutet, dass alle Elemente des Arrays um eine Position nach rechts verschoben werden. Das letzte Element wird dabei zum ersten. Mit `steps=2` wird alles um zwei Positionen nach rechts rotiert, usw. Eine Rotation nach links kann mit einer negativen Zahl für `steps` erreicht werden. Das folgende Beispiel ist der erste, einfache Test, den Sie in der Datei "BlackBoxTest.java" finden:

```
int[] values = new int[] { 1, 2 };
int[] expected = new int[] { 2, 1 };
BlackBox.rotateArray(values, 1);
assertArrayEquals(expected, values);
```

Dieser Test prüft, dass das Array `{ 1, 2 }` nach einer Rotation mit `steps=1` aussieht wie `{ 2, 1 }`. Die Methode `assertArrayEquals(expected, values)` prüft, dass die beiden Arrays `expected` und `values` die selben Elemente enthalten. Wenn nicht, schlägt der Test fehl.

Die genaue Definition von `rotateArray` lautet wie folgt: Nach einem Aufruf ist das Element am Index i gleich dem Element, das zuvor am Index $(i - \texttt{steps}) \mod \texttt{values.length}$ war, für alle i zwischen 0 und $\texttt{values.length} - 1$, inklusive. "mod" steht für *modulo* und bezeichnet den Rest einer Ganzzahl-Division (siehe Wikipedia). Mit diesem Wissen sollen Sie nun weitere Tests schreiben, die möglichst gut prüfen, ob sich eine Implementierung wunschgemäss verhält. Überlegen Sie sich genau, was für die Parameter `values` und `steps` angegeben werden kann, und wie `values` nach dem Aufruf von `rotateArray()` aussieht. Der gegebene Test prüft beispielsweise nur, dass bei einem Array mit zwei Elementen nach einer Rotation um `1` die Elemente vertauscht sind. Eine Implementierung, die ein Array einfach umkehrt, würde den Test auch bestehen.

Um die Stärke Ihrer Tests zu beurteilen, werden wir verschiedene, teilweise fehlerhafte Implementierungen mithilfe Ihrer Tests prüfen. Je mehr Fehler Ihre Tests aufdecken, desto besser. Tests sollten fehlschlagen, falls die Implementierung fehlerhaft ist, und erfolgreich durchlaufen, falls keine Fehler vorhanden sind.

Aufgabe 5: Levels (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich die allgemeinen Regeln.

Implementieren Sie die Methode `sumLevel(int[][] matrix, int level)` in der Klasse `Levels`, die für eine quadratische Matrix (Parameter `matrix`) und eine angegebene „Level"-Nummer (Parameter `level`) die Summe der Elemente dieses Levels berechnet.
Ein Level ist definiert durch die Schichten der Matrix, die von innen nach aussen gezählt werden. Level 0 entspricht dem Zentrum der Matrix. Höhere Level beschreiben jeweils die nächsten, äusseren Ränder der Matrix.

In Abbildung 2 finden Sie zwei Beispiele für Matrizen und ihre Levels. In Abbildung 2a beispielsweise besteht das innerste Level, Level 0, lediglich aus einem Element. Abbildung 2b zeigt eine Matrix mit drei Levels: Der äussere Rand der Matrix ist Level 2, der mittlere Rand Level 1 und Level 0 besteht aus den innersten vier Elementen.

1 Level: 1	2 Level: 1	3 Level: 1
4 Level: 1	5 Level: 0	6 Level: 1
7 Level: 1	8 Level: 1	9 Level: 1

(a) 3×3-Matrix

10 Level: 2	20 Level: 2	30 Level: 2	40 Level: 2	50 Level: 2	60 Level: 2
15 Level: 2	25 Level: 1	35 Level: 1	45 Level: 1	55 Level: 1	65 Level: 2
70 Level: 2	80 Level: 1	90 Level: 0	100 Level: 0	110 Level: 1	120 Level: 2
5 Level: 2	10 Level: 1	15 Level: 0	20 Level: 0	25 Level: 1	30 Level: 2
1 Level: 2	2 Level: 1	3 Level: 1	4 Level: 1	5 Level: 1	6 Level: 2
99 Level: 2	88 Level: 2	77 Level: 2	66 Level: 2	55 Level: 2	44 Level: 2

(b) 6×6-Matrix

Abbildung 2: Beispiele für gültige Matrizen und ihre Levels

Eine gültige Eingabe für den Parameter `matrix` ist ein `int`-Array, das eine quadratische Matrix der Grösse $n \times n$ mit $n > 0$ mit ausschliesslich positiven Einträgen repräsentiert. Die Methode `sumLevel` soll die Summe der Werte zurückgeben, die sich im angegebenen Level befinden.

Teilaufgabe 1 Implementieren Sie die Methode `sumLevel` so, dass der Wert `-2` zurückgegeben wird, falls die mit `matrix` übergebene Matrix keine gültige Eingabe ist oder `level` ausserhalb des gültigen Bereichs für Level-Nummern der Input-Matrix liegt. Beispielsweise soll `sumLevel` für die

Matrix in Abbildung 2b und Level 4 den Wert `-2` zurückgeben, da es in der Matrix kein Level 4 gibt.

Teilaufgabe 2 Erweitern Sie die Implementierung der Methode `sumLevel` nun mit der eigentlichen Funktionalität. Hier sind einige Beispiele:

- Für die Matrix in Abbildung 2a und Level 0 soll `sumLevel` 5 zurückgeben.
- Für die Matrix in Abbildung 2a und Level 1 soll `sumLevel` 40 zurückgeben.
- Für die Matrix in Abbildung 2b und Level 1 soll `sumLevel` 399 zurückgeben.