Meine Aufgaben Ratings:

- Best Case: Ihr macht alle Aufgaben (, wenn die Zeit reicht)
- Falls nicht: Ich mache Ratings zur Wichtigkeit der einzelnen Aufgaben

- (Keine offizielle Empfehlung, meine subjektive Meinung auf Basis meiner eigenen Erfahrung als Student in diesem Kurs)
- ! Sagen nichts über die Schwierigkeit der Aufgaben aus !

Wichtig für die Prüfung:
(Prüfungs- oder
prüfungsähnliche Aufgaben)

Wichtig für euer Verständnis:
(Aber nicht in Prüfungsform)

Wichtig für tiefes Verständnis/
Zusatzaufgaben:
(Bspw. viele vom gleichen Typ)

Aufgabe 1: Sieb des Eratosthenes

Schreiben Sie ein Programm "Sieb.java", das eine Zahl *limit* einliest und die Anzahl der Primzahlen, die grösser als 1 und kleiner oder gleich dem *limit* sind, ausgibt. Dazu ermitteln Sie in einem ersten Schritt alle Primzahlen, die kleiner oder gleich *limit* sind. Dieses Teilproblem können Sie mit dem Sieb des Eratosthenes lösen. Das Sieb des Eratosthenes findet Primzahlen bis n. Man betrachtet alle Zahlen von 2 bis n und streicht zuerst alle Vielfachen der ersten Zahl (2). Dann geht man zur nächsten ungestrichenen Zahl (3) und wiederholt das Streichen ihrer Vielfachen. Das macht man, bis man dies für alle Zahlen gemacht hat. Sie können ein `Boolean`-Array verwenden, um zu speichern, welche Zahlen Primzahlen sind und welche nicht. Übrig bleiben die Primzahlen. Danach können Sie die Anzahl der gefundenen Primzahlen anhand dieses Arrays bestimmen.

Beispiel: Für $limit = 13$ sollte Ihr Programm 6 ausgeben (Primzahlen: 2, 3, 5, 7, 11, 13).

Hinweis: Es ist nicht zwingend nötig von 2 bis n zu gehen. Von 2 bis $\sqrt{n}$ zu gehen reicht bereits aus, da eine Zahl $\leq n$ nicht einen Teiler grösser als $\sqrt{n}$ ausser sich selbst haben kann.

Aufgabe 2: Arrays

1. Implementieren Sie die Methode `ArrayUtil.zeroInsert(int[] x)` in der Datei "ArrayUtil.java". Die Methode nimmt einen Array `x` als Argument und gibt einen Array zurück. Der zurückgegebene Array soll die gleichen Werte wie `x` haben, ausser: Wenn eine positive Zahl direkt auf eine negative Zahl folgt oder wenn eine negative Zahl direkt auf eine positive Zahl folgt, dann wird dazwischen eine 0 eingefügt.

   Beispiele:

   - Wenn `x` gleich `[3, 4, 5]` ist, dann wird `[3, 4, 5]` zurückgegeben.
   - Wenn `x` gleich `[3, 0, -5]` ist, dann wird `[3, 0, -5]` zurückgegeben.
   - Wenn `x` gleich `[-3, 4, 6, 9, -8]` ist, dann wird `[-3, 0, 4, 6, 9, 0, -8]` zurückgegeben.

   Versuchen Sie, die Methode rekursiv zu implementieren.

Aufgabe 2: Arrays

2. Implementieren Sie die Methode `ArrayUtil.tenFollows(int[] x, int index)`. Die Methode gibt einen Boolean zurück. Die Methode soll `true` zurückgeben, wenn im Array `x` ab Index `index` der zehnfache Wert einer Zahl `n` direkt der Zahl `n` folgt. Dies muss nur für das erste Auftreten der Zahl `n` ab Index `index` im Array `x` geprüft werden. Ansonsten soll die Methode `false` zurückgeben.

   Beispiele:

   - `tenFollows([1, 2, 20], 0)` gibt `true` zurück.
   - `tenFollows([1, 2, 7, 20], 0)` gibt `false` zurück.
   - `tenFollows([3, 30], 0)` gibt `true` zurück.
   - `tenFollows([3], 0)` gibt `false` zurück.
   - `tenFollows([1, 2, 20, 5], 1)` gibt `true` zurück.
   - `tenFollows([1, 2, 20, 5], 2)` gibt `false` zurück.

   Die `main` Methode in `ArrayUtil` gibt die oben genannten Beispielaufrufe sowie das entsprechende Ergebnis der jeweiligen Methode aus. Hiermit können Sie überprüfen, ob Ihre Implementierungen die richtigen Ergebnisse zurückliefern. In "ArrayUtilTest.java" im Ordner "test" in der Übungsvorlage finden Sie zusätzlich einige Unit-Tests für beide Methoden (für eine detaillierte Beschreibung zu automatisiertem Testen und der Ausführung solcher Tests siehe Aufgabe 3). Sie können die `main` Methode und die Tests beliebig abändern und/oder mit Ihren eigenen Inputs erweitern.

Aufgabe 3: 2D Arrays

Gegeben einer Matrix M, prüfen Sie zuerst ob diese eine $n \times n$ Matrix ist, deren Elemente positive ganze Zahlen sind. Danach prüfen Sie ob zusätzlich alle Zahlen kleiner gleich n^2 sind. Somit gilt nun $0 < m_{i,j} \leq n^2$. Prüfen Sie ebenfalls, ob die Elemente der Matrix jeweils genau einmal vorkommen, sprich ob $m_{x,y} = m_{p,q} \Rightarrow (x = p) \wedge (y = q)$ gilt. Wir sagen, dass die Matrix M *perfekt* ist, wenn zusätzlich alle Zeilensummen und Spaltensummen gleich sind (also $\sum_{k=0}^{k=n-1} m_{i,k} = \sum_{k=0}^{k=n-1} m_{j,k}$ für alle i, j und $\sum_{k=0}^{k=n-1} m_{k,i} = \sum_{k=0}^{k=n-1} m_{k,j}$ für alle i, j mit $0 \leq i, j < n$).

Vervollständigen Sie die Methode `boolean checkMatrix(int[][] m)` von der Klasse Matrix, so dass diese Methode `true` zurückgibt wenn die Input Matrix *perfekt* ist, und `false` sonst. Sie können davon ausgehen, dass der Parameter `m` nicht `null` ist. Alle anderen Eigenschaften müssen Sie selber testen. Eine Matrix ist nur perfekt, wenn alle genannten Eigentschaften gelten.

Testen Sie Ihr Programm ausgiebig - am besten mit JUnit - und pushen Sie die Lösung vor dem Abgabetermin. Wir haben Ihnen einen JUnit Test in der Klases `MatrixTest` bereits erstellt.

Aufgabe 4: Testen mit JUnit

Zweck des Programms:

- Wochentag eines Datums (nach 01.01.1900) ausgeben
  Beispiel:13.10.2017 → Friday
  Gibt fälschlicherweise aber *“The 13.10.2017 is a Sunday”* aus.
- Berücksichtigt Schaltjahre (“Leap year”)

Funktionsweise:

1. Überprüft, ob Datum OK ist
2. Zählt die Tage ab 1.1.1900 bis zum eingegebenen Datum
3. Wochentag = Tage % 7

Aufgabe 4: Testen mit JUnit

Tests in *PerpetualCalendarTest.java*

- Einzelne Tests prüfen Rückgabewerte von einzelnen Methoden des Programms *PerpetualCalendar.java*
- Tests sollten *interessante* Parameter für die Methoden testen
- Beispiel `testCountDaysInYear()`:
  ```
  assertEquals(366, PerpetualCalendar.countDaysInYear(1904));
  ```
  1904 ist ein Schaltjahr, also sollte `countDaysInYear()` 366 Tage zurückgeben

DEMO

Aufgabe 5: Matching Numbers

Implementieren Sie die Methode `Match.matchNumber(long A, int M)`. Die Methode soll für eine Zahl `A` und eine nicht-negative drei-stellige Zahl `M` die Position von `M` in `A` zurückgeben. Sei `M` eine Zahl mit den Ziffern $M_2M_1M_0$ (das heisst, es gilt $M = M_0 + 10 \cdot M_1 + 100 \cdot M_2$), wobei jede Ziffer 0 sein kann. Zusätzlich sei `A` eine Zahl, sodass A_i die i-te Ziffer von `A` ist (das heisst, es gilt $|A| = \sum_i 10^i \cdot A_i$), wobei `A` unendlich viele führende Nullen hat. Die Position von `M` in `A` ist die kleinste Zahl j, sodass $A_j = M_0$ und $A_{j+1} = M_1$ und $A_{j+2} = M_2$ gilt. Die Methode soll `-1` zurückgeben, falls es kein solches j gibt.

Beispiele:

`matchNumber(32857890, 789)` soll `1` zurückgeben.
`matchNumber(37897890, 789)` soll `1` zurückgeben.
`matchNumber(1800765, 7)` soll `2` zurückgeben.
`matchNumber(1800765, 8)` soll `-1` zurückgeben (die drei Ziffern von `8` sind `008`).
`matchNumber(75, 7)` soll `1` zurückgeben (da `007` and Position `1` von `0075` ist).

Aufgabe 5: Matching Numbers

Implementieren Sie die Berechnung in der Methode `int matchNumber(long A, int M)`, welche sich in der Klasse `Match` befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass $0 \leq \texttt{M} < 1000$ gilt.

In der `main` Methode der Klasse `Match` finden Sie die oberen Beispiele als kleine Tests, welche Beispiel-Aufrufe zur `matchNumber`-Methode machen und welche Sie als Grundlage für weitere Tests verwenden können. In der Datei `MatchTest.java` geben wir die gleichen Tests zusätzlich auch als JUnit Test zur Verfügung. Sie können diese ebenfalls nach belieben ändern. Es wird *nicht* erwartet, dass Sie für diese Aufgabe den JUnit Test verwenden.

Tipp: Die Methode `Integer.toString(int i)` wandelt einen Integer in einen String um.

Aufgabe 6: Substring-Counter (Bonus!)

- Diese Aufgabe gibt Bonuspunkte!
- Regeln findet ihr hier: https://lec.inf.ethz.ch/infk/eprog/2025/exercises/additionals/bonusaufgaben_regeln.pdf