

252-0027-00: Einführung in die Programmierung

All Bonus Tasks

Inhaltsverzeichnis

1	Matching Numbers (2023 W4)	4
2	Close Neighbors (2022 W4)	4
3	Max3 (2021 W4)	5
4	Median (2020 W4)	5
5	Ohne Sieben (2019 W4)	6
6	String-Addition (2018 W4)	6
7	Matrixmuster (2023 W5)	7
8	Wellen (2022 W5)	8
9	Schneckenkonstruktion (2021 W5)	9
10	Die perfekte Matrix (2020 W5)	10
11	Hotellerie (2019 W5)	10
12	Gerätevermietung (2018 W5)	12
13	Dreiecksmatrix (2023 W6)	13
14	Prefixkonstruktion (2022 W6)	14
15	Split57 (2021 W6)	15
16	Familienbeziehungen (2020 W6)	15
17	Schweizer Uhrzeit (2019 W6)	16
18	Talsohlen (2018 W6)	17

19 Rechnungen (2023 W7)	18
20 Roulette (2022 W7)	20
21 Grid (2021 W7)	21
22 Subtraktion von Listen (2020 W7)	22
23 Split (2019 W7)	23
24 Verzahnungen (2018 W7)	24
25 Graphgenerierung (2023 W8)	25
26 Executable Graph (2022 W8)	25
27 Labyrinth (2021 W8)	27
28 Mindestanzahl an Teilfolgen (2020 W8)	27
29 Enthalten mit Abstand (2019 W8)	28
30 Umkehrung (2018 W8)	28
31 Square Grid (2023 W9)	29
32 Klassen Rätsel (2022 W9)	30
33 Ballspiel (2021 W9)	31
34 Klassen Rätsel (2020 W9)	32
34.1 Anhang: Testprogramm Bonusaufgabe	32
34.2 Anhang: Klassen Bonusaufgabe	33
34.3 Anhang: Ausgabe Bonusaufgabe	34
35 Klassen Rätsel (2019 W9)	34
35.1 Anhang: Testprogramm Bonusaufgabe	34
35.2 Anhang: Ausgabe Bonusaufgabe	36
36 Klassen Rätsel (2018 W9)	36
36.1 Testprogramm Bonusaufgabe	36
36.2 Ausgabe Bonusaufgabe	37
37 Datenbanken (2023 W10)	38
38 Equivalent Executable Graphs (2022 W10)	40
39 Flex Array (2021 W10)	40

40 Pyramide (2020 W10)	42
41 Wahlen (2019 W10)	43
42 Mini-Taschenrechner (2018 W10)	44
42.1 Testprogramm Bonusaufgabe	46
43 Contact Tracing (2023 W11)	47
44 Biome (2022 W11)	49
45 Hamster Kartell (2021 W11)	51
46 Hogwarts (2020 W11)	52
47 Notenauswertung (2019 W11)	54
48 Bienen Syndikat (2018 W11)	55
49 Datenbanken (2023 W12)	56
50 Konstruktion (2022 W12)	57
51 Netzwerk (2021 W12)	61
52 Generische Listen (2020 W12)	63
52.1 MyList Interface	63
52.2 MyListNode Interface	64
53 Wörter (2019 W12)	65
54 Palindrome (2018 W12)	66

Week 4

Matching Numbers (2023 W4)

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).

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 $0 \leq M < 1000$ 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.

Close Neighbors (2022 W4)

Schreiben Sie ein Programm, welches für eine sortierte Folge X von `int`-Werten (x_1, x_2, \dots, x_n) und einen `int`-Wert `key` die drei unterschiedlichen Elemente x_a , x_b und x_c aus X zurückgibt, die dem Wert `key` am nächsten sind. Für x_a , x_b und x_c muss gelten, dass $|\text{key} - x_a| \leq |\text{key} - x_b| \leq |\text{key} - x_c| \leq |\text{key} - x_i|$ für alle $i \neq a, b, c$ und dass $x_a \neq x_b \neq x_c \neq x_a$. Wenn die drei Werte nicht eindeutig bestimmt sind, dann ist jede Lösung zugelassen, die die obige Bedingung erfüllt.

Beispiele:

Die nächsten Nachbarn für `key == 5` in $(1, 4, 5, 7, 9, 10)$ sind 5, 4, 7.

Die nächsten Nachbarn für `key == 5` in $(1, 4, 5, 6, 9, 10)$ sind 5, 4, 6 oder 5, 6, 4.

Die nächsten Nachbarn für `key == 10` in $(1, 4, 5, 6, 9, 10)$ sind 10, 9, 6.

Implementieren Sie die Berechnung in der Methode `int[] neighbor(int[] sequence, int key)`, welche sich in der Klasse `Neighbor` befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass das Argument `sequence` nicht null ist, sortiert ist, nur

unterschiedliche Elemente enthält, und mindestens drei Elemente enthält. Denken Sie daran, dass der Wert `key` nicht unbedingt in der Folge `X` auftritt. Sie dürfen den Eingabearray `input` nicht ändern.

In der `main` Methode der Klasse `Neighbor` finden Sie die oberen Beispiele als kleine Tests, welche Beispiel-Aufrufe zur `neighbor`-Methode machen und welche Sie als Grundlage für weitere Tests verwenden können. In der Datei `NeighborTest.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.

Max3 (2021 W4)

Implementieren Sie die Methode `Max3.max3(int[] x)` in der Datei `"Max3.java"`. Die Methode nimmt einen Array `x` als Argument und soll die drei grössten Werte aus `x` in einem Array zurückgeben. Der zurückgegebene Array soll aufsteigend sortiert sein (der kleinste Wert beim Index 0 und der grösste Wert beim Index 2). Die Methode darf den Inhalt vom Argument `x` nicht ändern. Sie dürfen annehmen, dass das Argument `x` nicht `null` ist und mindestens drei Werte enthält. Das heisst, dass wir bei der Korrektur die Methode weder mit `null` noch mit Arrays mit weniger als drei Werten aufrufen. Die Deklaration der Methode ist bereits vorgegeben. Wie in den Regeln festgehalten, solange nicht explizit erlaubt, dürfen Sie die Methodendeklaration nicht ändern. **Committen und pushen Sie Ihre Lösung!**

Beispiele:

- `max3([1, 2, 3, 4, 5])` gibt `[3, 4, 5]` zurück.
- `max3([4, 2, 1, 5, 3])` gibt `[3, 4, 5]` zurück.
- `max3([5, 4, 1, 5, 3])` gibt `[4, 5, 5]` zurück.
- `max3([2, 1, 20])` gibt `[1, 2, 20]` zurück.
- `max3([1, 4, -2, 3])` gibt `[1, 3, 4]` zurück.

In der `main` Methode in der Datei `"Max3.java"` werden die oberen Beispiele aufgerufen und geprüft. Sie können das als Grundlage für weitere Tests verwenden.

In der Datei `"Max3Test.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.

Median (2020 W4)

Schreiben Sie ein Programm, welches den Median einer Folge von `int`-Werten (x_1, x_2, \dots, x_n) berechnet und zurückgibt. Der Median ist definiert als der Wert, der sich in der Mitte der sortierten Liste dieser Zahlen befindet. Falls die Anzahl Werte gerade ist, entspricht der Median dem arithmetischen Mittel der beiden am nächsten bei der Mitte liegenden Zahlen. In diesem Fall ist der Median nicht zwingend eine ganze Zahl.

Beispiele:

Der Median von (1, 5, 4, 3, 0) ist 3.

Der Median von (1000, -100, 0) ist 0.

Der Median von (4, 17, 5, 1) ist 4.5.

Implementieren Sie die Berechnung in der Methode `double median(int[] sequence)`, welche sich in der Klasse `Median` befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass das Argument `sequence` nicht `null` ist und mindestens ein Element enthält.

Tip: Sie müssen das Sortieren eines Arrays nicht selber implementieren. **Committen und pushen Sie Ihre Lösung!**

In der `main` Methode der Klasse `Median` finden Sie die oberen Beispiele als kleine Tests, welche Beispiel-Aufrufe zur `median`-Methode machen und welche Sie als Grundlage für weitere Tests verwenden können. In der Datei `MedianTest.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.

Ohne Sieben (2019 W4)

1. Ihre Aufgabe ist eine nicht-negative Zahl $N \geq 0$ in zwei Summanden s_1 und s_2 zu zerlegen, so dass $s_1 \geq s_2$ und $s_1 + s_2 = N$ gilt und die Ziffer 7 weder in s_1 noch in s_2 auftritt (in der Dezimaldarstellung). Zum Beispiel ist für $N = 9743$ eine Zerlegung in $s_1 = 6852$ und $s_2 = 2891$ eine mögliche Lösung.

Vervollständigen Sie dafür in der Klasse `OhneSieben` die Methode `ohneSieben()`. Die Methode nimmt einen Parameter `zahl`, welcher N entspricht, und gibt einen Array mit Länge 2 zurück, in welchem der Wert beim Index 0 der Zahl s_1 und beim Index 1 der Zahl s_2 entspricht.

2. Committen und pushen Sie Ihre Lösung!

Tip: Zum Testen können Sie die `main`-Methode verwenden. Wir haben für Sie 3 Tests vorgeschrieben, welche Ergebnisse mit der Methode `richtigesResultat()` prüfen. **Achtung:** Die Methode `richtigesResultat()` überprüft nicht, ob die beiden Summanden die Ziffer 7 nicht enthalten. Wenn Sie wollen, dann können Sie die Methode entsprechend anpassen. Beachten Sie, dass Ihre Lösung mehrere Tests hintereinander bestehen können muss.

String-Addition (2018 W4)

1. Gegeben seien zwei Strings, genannt s_1 und s_2 , der Länge 4. Jeder String hat entweder das Format `+ABC` oder `-ABC`, wobei A, B und C Ziffern zwischen 0 und 9 sind. Eine oder mehrere Nullen am Anfang eines Strings sind erlaubt, d.h. `+012` oder `-001` sind erlaubte Strings. Ihre Aufgabe ist es, die Zahlen, welche durch s_1 und s_2 dargestellt werden, zu addieren und im selben Format (`+ABC` oder `-ABC`) wieder zurück zu geben. (Sie können davon ausgehen, dass sich die Summe $s_1 + s_2$ wieder in diesem Format darstellen lässt.)

In der Klasse `StringAddition` finden Sie die Methode `add()`, welche zwei Parameter `s1` und `s2` hat, welche den beiden Strings `s1` und `s2` entsprechen. Ändern Sie die Methode so ab, dass sie, wie oben beschrieben, die Summe berechnet und als String zurückgibt.

Wichtig: Ihre Implementierung der Methode `add` darf nur Addition, Subtraktion und Vergleiche verwenden; Sie dürfen also *nicht* Multiplikation oder Division verwenden (irgendwo in der Methode). Zusätzlich sind jegliche nicht selber geschriebene Methoden verboten, welche zwischen String und Integer konvertieren (zum Beispiel `Integer.parseInt()`). Ausserdem dürfen Sie weder den Namen der Klasse `StringAddition`, noch den Namen, die Parameter oder den Rückgabewert der `add()`-Methode verändern.

2. Committed und pushen Sie Ihre Lösung!

Tip: Zum Testen können Sie die `main`-Methode verwenden, welche 2 Strings einliest, `add()` aufruft, und die von `add()` berechnete Summe ausgibt.

Week 5

Matrixmuster (2023 W5)

Gegeben sei eine $k \times k$ -Matrix M von `int`-Werten, welche in Java als ein 2D-Array repräsentiert wird. Diese Matrix enthält das Muster, das Sie in einer $n \times n$ Matrix O finden sollen ($n \geq k > 0$). Wir sagen, dass die Matrix O die Matrix M für eine Ursprungsposition (x, y) (mit $0 \leq x < n$, $0 \leq y < n$) enthält, wenn für alle i und j mit $0 \leq i < k$, $0 \leq j < k$ gilt: $O[x+i][y+j] = M[i][j]$. Es ist möglich, dass eine Matrix O die Matrix M für verschiedene Ursprungspositionen $(x_1, y_1), (x_2, y_2), \dots$ enthält (oder für keine).

Wenn es keine Ursprungsposition (x, y) gibt, für die eine Matrix O die Matrix M enthält, dann kann durch maximal k^2 Korrekturschritte die Matrix O in eine Matrix O' so verändert werden, so dass danach die Matrix O' die Matrix M enthält. (Es müssen für eine Ursprungsposition (x, y) alle die Elemente $O[x+i][y+j]$ geändert werden, für die $O[x+i][y+j] \neq M[i][j]$. Jeder Korrekturschritt setzt *ein* Element des Arrays.)

Implementieren Sie in der Klasse `Pattern` die Methode `match(int[][] origin, int[][] muster)`, welche die minimale Anzahl der Korrekturschritte und eine dazu passende Ursprungsposition findet. Dabei ist `muster` eine $k \times k$ Matrix M von `int`-Werten und `origin` eine $n \times n$ Matrix von `int`-Werten. Sie können davon ausgehen, dass beide Parameter nicht null sind und dass $k \leq n$.

Die Methode `match` gibt das Ergebnis in einem Objekt `record` der Klasse `MatchRecord` zurück. Dabei muss (`record.x`, `record.y`) eine Ursprungsposition in der Matrix `origin` sein, für welche die Anzahl der Korrekturschritte minimal ist, damit die korrigierte Matrix `origin'` die Matrix `muster` enthält. `record.count` muss dabei der minimalen Anzahl Korrekturschritte entsprechen. Wenn die Matrix `origin` die Matrix `muster` an der Position (x, y) ohne Korrekturschritte enthält, dann ist die Anzahl der Korrekturschritte 0.

Wenn es mehrere Positionen $A = \{(x_1, y_1), (x_2, y_2), \dots\}$ gibt, so dass die Matrix `origin` die Matrix `muster` für alle Ursprungspositionen $(x_i, y_i) \in A$ mit q Korrekturschritten enthält und q das Minimum der nötigen Korrekturschritte ist, dann können Sie die Anzahl Korrekturschritte q und eine beliebige Position $(x_i, y_i) \in A$ zurückgeben.

In der Klasse `Pattern` finden Sie eine `main` Methode, welche Sie verwenden können, um Ihre Implementierung zu testen. Tests finden Sie in der Datei `"PatternTest.java"`. Die Datei `"Grading-PatternTest.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.

Wellen (2022 W5)

Sei M eine $n \times n$ Matrix mit $n > 0$. Ein Index $x = (i, j)$ von M ist ein Paar von einer Zahl i für die Zeile und einer Zahl j für die Spalte mit $i, j \in \{0, \dots, n-1\}^2$, welche eine Position in der Matrix M beschreibt, wobei M_x der Wert von M beim Index x ist. Zum Beispiel, die Matrix in der Figur 1 hat Wert 6 beim Index $(2, 0)$. Wir definieren ein Set von Startindizes (START), Endindizes (END), und Nachbarindizes (NEIGHBOR_x für einen Index x) wie folgt:

$$\begin{aligned}\text{START} &= \{(i, j) \mid i = 0 \vee j = 0\} \cap \{0, \dots, n-1\}^2 \\ \text{END} &= \{(i, j) \mid i = n-1 \vee j = n-1\} \cap \{0, \dots, n-1\}^2 \\ \text{NEIGHBOR}_{(i,j)} &= \{(i+1, j), (i, j+1), (i-1, j), (i, j-1)\} \cap \{0, \dots, n-1\}^2\end{aligned}$$

Ein Sequenz von unterschiedlichen Indexen $X = x_0, \dots, x_s$ mit $s \geq 0$ beschreibt eine Welle in M , falls:

1. X mit einem Startindex anfängt ($x_0 \in \text{START}$) und einem Endindex aufhört ($x_s \in \text{END}$), wobei x_0 gleich x_s sein kann
2. Aufeinanderfolgende Indexe in X Nachbarindexe sind: $\forall i. 0 \leq i < s \Rightarrow x_{i+1} \in \text{NEIGHBOR}_{x_i}$
3. Für jeden Index maximal zwei Nachbarn in X sind: $\forall x \in X. |\text{NEIGHBOR}_x \cap X| \leq 2$
4. M bei jedem Index in X den gleichen Wert hat: $\exists v. \forall x \in X. M_x = v$
5. M bei jedem Index, welcher nicht in X ist, aber ein Nachbarindex eines Indexes aus X ist, einen kleineren Wert hat: $\forall x \in X, y \in (\text{NEIGHBOR}_x \setminus X). M_y < M_x$

Die Matrix M enthält k Wellen, falls es k unterschiedliche Sequenzen gibt (das heisst, dass keine Sequenzen identisch sind), welche Wellen in M beschreiben. Zum Beispiel, die Matrix in Figur 1 enthält drei Wellen, welche mit blau markiert sind. Eine der Wellen wird durch die Sequenz $(0, 4), (1, 4), (2, 4), (2, 5)$ beschrieben, bei welchen die Matrix überall den Wert 4 hat.

6	6	7	3	4	2
6	2	7	1	4	3
6	6	7	2	4	4
4	5	7	7	2	3
9	9	1	7	7	4
7	9	4	5	7	4

Abbildung 1: Ein Beispiel mit 3 Wellen. Die Wellen sind mit blau markiert.

1	2	3
4	5	6
7	8	9

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Abbildung 2: Zwei Beispiele.

Implementieren Sie die Methode `Waves.waves(int[][] matrix)`, welche eine quadratische Matrix als Argument nimmt und die maximale Anzahl Wellen in `matrix` zurückgibt. Sie dürfen annehmen, dass für das Argument alle unterschiedlichen Sequenzen, welche Wellen beschreiben, disjunkt sind (das heisst, dass die Sequenzen kein gemeinsames Element enthalten). Die Datei `“WavesTest.java”` enthält bereits einige Tests. Wir empfehlen die Tests anzusehen, da die Tests einige Beispiele zeigen. Testen Sie Ihr Programm ausgiebig—am besten mit JUnit—and pushen Sie die Lösung vor dem Abgabetermin. **Tipp:** Überprüfen Sie für jeden Index in `START`, ob er der Anfang einer Welle ist, indem Sie über die mögliche Welle iterieren.

Schneckenkonstruktion (2021 W5)

Sei M eine $n \times n$ Matrix mit $n > 0$. Wir teilen M in $\lceil \frac{n}{2} \rceil$ Ringe. Ein Element $m_{i,j}$ von M ist auf Ring $k \in \{0, \dots, \lceil \frac{n}{2} \rceil - 1\}$, genau dann wenn i oder j gleich k oder $n - k - 1$ ist und sowohl i als auch j zwischen k und $n - k - 1$ sind (das heisst, es gilt $\{i, j\} \cap \{k, n - k - 1\} \neq \emptyset \wedge \{i, j\} \subseteq \{k, \dots, n - k - 1\}$). Die Elemente eines Rings haben eine feste Reihenfolge. Das erste Element von Ring k ist $m_{k,k}$ und die restlichen Elemente folgen im Uhrzeigersinn. Zum Beispiel, die linke Matrix in Figur 2 hat zwei Ringe. Ring 0 hat die Elemente 1, 2, 3, 6, 9, 8, 7, 4 (in dieser Reihenfolge) und Ring 1 hat das Element 5. Die rechte Matrix hat ebenfalls zwei Ringe. Ring 0 hat die Elemente 1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5 und Ring 1 hat die Elemente 6, 7, 11, 10. Eine *Schneckenkonstruktion* S einer Matrix M ist der kleinste 2-dimensionale Array, bei welchem $s_{k,l}$ das l -te Element vom Ring k von M enthält. Die Matrix M ist die *ursprüngliche* Matrix der Schneckenkonstruktion S .

- Implementieren Sie die Methode `Matrix.toSnail(int[][] matrix)`. Die Methode nimmt als Argument eine Matrix (gespeichert in einem 2-dimensionalen Array) und gibt die Schneckenkonstruktion des Arguments zurück. Sie dürfen annehmen, dass das Argument nicht null ist und eine korrekte $n \times n$ Matrix mit $n > 0$ ist.
- Implementieren Sie die Methode `Matrix.fromSnail(int[][] snailConstruction)`. Die Methode nimmt als Argument eine Schneckenkonstruktion und gibt die ursprüngliche Matrix zurück. Sie dürfen annehmen, dass das Argument nicht null ist und eine korrekte Schneckenkonstruktion einer ursprünglichen $n \times n$ Matrix mit $n > 0$ ist.
- Implementieren Sie die Methode `Matrix.areRowsPositive(int[][] snailConstruction)`. Die Methode nimmt eine Schneckenkonstruktion als Argument und gibt `true` zurück, genau dann wenn die ursprüngliche Matrix keine Zeile hat, bei welcher die Summe der Elemente nicht positiv ist. Sie dürfen annehmen, dass das Argument nicht null ist und eine korrekte Schneckenkonstruktion einer ursprünglichen $n \times n$ Matrix mit $n > 0$ ist.

- d) Implementieren Sie die Methode `Matrix.areRingsDense(int[] [] matrix)`. Die Methode nimmt eine $n \times n$ Matrix als Argument und gibt `true` zurück, genau dann wenn die Matrix keinen Ring hat, bei welchem mehr als ein viertel der Elemente des Rings 0 sind (ein Ring mit c Elementen, darf maximal $\lfloor \frac{c}{4} \rfloor$ mal 0 als Element haben). Die dürfen annehmen, dass das Argument nicht `null` ist und eine korrekte $n \times n$ Matrix mit $n > 0$ ist.

Die Datei “MatrixTest.java” enthält bereits einige Tests. Wir empfehlen die Tests anzusehen, da die Tests einige Beispiele für alle Aufgaben zeigen. Testen Sie Ihr Programm ausgiebig—am besten mit JUnit—und pushen Sie die Lösung vor dem Abgabetermin.

Die perfekte Matrix (2020 W5)

Sei M eine $n \times n$ Matrix deren Elemente positive ganze Zahlen sind und für die gilt $0 < m_{i,j} \leq n^2$ und $m_{x,y} = m_{p,q} \Rightarrow (x = p) \wedge (y = q)$. 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 Eigenschaften 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 Klasse `MatrixTest` bereits erstellt.

Hotellerie (2019 W5)

In dieser Aufgabe analysieren Sie die Zimmerbuchungsdaten eines Hotels innerhalb eines Kalenderjahrs. Ein Beispiel für die Zimmerbuchungsdaten finden Sie in der Datei “hotelDaten1.txt”. Die Datei enthält eine Liste der Buchungen, welche das Hotel getätigt hat. In der ersten Zeile steht nur die Anzahl Buchungen. Für jede Zimmerbuchung gibt es genau eine weitere Zeile, in der Sie (durch Leerzeichen getrennt) folgende Informationen finden (in dieser Reihenfolge):

1. Die Nummer des Zimmers (eine positive ganze Zahl zwischen 1 und 256 inklusive).
2. Den Tag im Jahr, an dem die Buchung des Zimmers begonnen hat (eine positive ganze Zahl zwischen 1 und 366 inklusive).
3. Den Tag im Jahr, an dem die Buchung des Zimmers geendet hat (eine positive ganze Zahl zwischen 1 und 366 inklusive).
4. Der Preis des Zimmers pro angefangenem Tag (Eine nicht negative reelle Zahl).
5. Der Discount des Zimmers (eine ganze Zahl zwischen 0 und 100, welche den Discount als Prozent angibt: Zum Beispiel, 0 ist kein Discount, 50 ist halber Preis und 100 heisst, dass das Zimmer Gratis ist).

Alle Buchungen enden im gleichen Jahr, da das Hotel an Sylvester geschlossen hat. Es gibt für jede Zimmernummer ein Zimmer, auch wenn dieses in einem Jahr eventuell nicht gebucht wurde und daher nicht in der Datei erscheint. Ein Zimmer kann an einem Tag höchstens einmal gebucht werden; das gilt auch für den Tag, an dem die Buchung endet. Es gibt keine angebrochenen Tage, Zimmer werden immer für ganze Tage vermietet.

Ihr Programm soll die Datei einmal einlesen und anschliessend einige Fragen beantworten. Dazu vervollständigen Sie die Methode `analyse()` in der Klasse `Hotellerie`. Die Methode hat zwei Argumente: ein `input`-Argument zum Lesen der Datei und ein `output`-Argument zum Ausgeben der Antworten. Sie sollen also vom `input` lesen und Antworten zu den folgenden Fragen auf dem `output` ausgeben:

1. Welches Zimmer wurde am häufigsten ausgeliehen (d.h. die grösste Anzahl an Buchungen)? Geben Sie die Zimmernummer aus, mit diesem `println()`-Statement (ändern Sie nicht den Text):

```
output.println("Am haeufigsten gebucht: "+ zimmerNummer);
```

2. Welches Zimmer wurde am meisten besetzt (d.h. für die grösste Gesamtanzahl an Tagen)? Geben Sie die Zimmernummer aus, mit diesem `println()`-Statement (ändern Sie nicht den Text):

```
output.println("Am meisten besetzt: "+ zimmerNummer);
```

3. Welches Zimmer hat dem Hotel den grössten Betrag eingebracht? Geben Sie die Zimmernummer aus, mit diesem `println()`-Statement (ändern Sie nicht den Text):

```
output.println("Groessten Betrag eingebracht: "+ zimmerNummer);
```

4. Was sind die Gesamteinnahmen des Hotels in diesem Jahr? Geben Sie den Betrag aus, mit diesem `println()`-Statement (ändern Sie nicht den Text):

```
output.println("Gesamteinnahmen des Hotels: "+ summe);
```

Jede der Informationen 1.–4. soll (in dieser Reihenfolge) auf einer separaten Zeile ausgegeben werden. Beträge werden als ungerundete reelle Zahlen ausgegeben. Zimmernummern werden als natürliche Zahlen ausgegeben. Falls bei einer Information mehrere gleichwertige Ausgaben möglich sind, kann Ihr Programm *eine* beliebige der gleichwertigen Lösungen ausgeben.

Ihr Programm muss nur wohlgeformte, nicht-leere Eingabe-Dateien unterstützen. Ein Beispiel einer solchen Datei finden Sie im Projekt unter dem Namen `hotelDaten1.txt`. Exceptions im Zusammenhang mit Ein- und Ausgabe müssen nicht behandelt werden. Für die Beispiel-Datei sollte die Ausgabe wie folgt aussehen:

```
Am haeufigsten gebucht: 1
Am meisten besetzt: 42
Groessten Betrag eingebracht: 256
Gesamteinnahmen des Hotels: 22000.75
```

Testen Sie Ihr Programm ausgiebig—am besten mit JUnit—und pushen Sie die Lösung vor dem Abgabetermin. Wir haben Ihnen einen JUnit Test bereits erstellt.

Gerätevermietung (2018 W5)

In dieser Aufgabe analysieren Sie die Vermietungsdaten einer Firma für Elektrogeräte, innerhalb eines Kalenderjahrs. Ein Beispiel für die Vermietungsdaten finden Sie in der Datei "vermietungen.txt". Die Datei enthält eine Liste der Mietvorgänge, welche die Firma getätigt hat. Für jede Vermietung gibt es genau eine Zeile, in der Sie (durch Leerzeichen getrennt) folgende Informationen finden:

1. Die Artikelnummer des Geräts (eine positive ganze Zahl zwischen 1 und 1000 inklusive).
2. Den Tag im Jahr, an dem der Artikel gemietet wurde (eine positive ganze Zahl zwischen 1 und 366 inklusive).
3. Den Tag im Jahr, an dem der Artikel zurückgegeben wurde (eine positive ganze Zahl zwischen 1 und 366 inklusive).

Alle Geräte werden im gleichen Jahr zurück gegeben, in dem sie auch gemietet wurden. Es gibt für jede Artikelnummer ein Gerät, auch wenn dieses in einem Jahr eventuell nicht ausgeliehen wurde und daher nicht in der Datei erscheint. Ein Gerät kann an einem Tag höchstens einmal vermietet werden; das gilt auch für den Tag, an dem es zurückgegeben wurde. Es gibt keine angebrochenen Tage, Geräte werden immer für ganze Tage vermietet.

Ihr Programm soll die Datei einmal einlesen und anschliessend einige Fragen beantworten. Dazu vervollständigen Sie die Methode `analyse()` in der Klasse `Vermietung` im Projekt "Aufgabe 4". Die Methode hat zwei Argumente: ein `input`-Argument zum Lesen der Datei und ein `output`-Argument zum Ausgeben der Antworten. Sie sollen also vom `input` lesen und Antworten zu den folgenden Fragen auf dem `output` ausgeben:

1. Welches Gerät wurde am längsten ausgeliehen (d.h. für die grösste Gesamtanzahl an Tagen)? Geben Sie die Artikelnummer aus, mit diesem `println()`-Statement (ändern Sie nicht den Text):

```
output.println(Laengste Zeit: "+ artikelNummer);
```

2. Welches Gerät wurde am häufigsten ausgeliehen (d.h. die grösste Anzahl an Vermietungen)? Geben Sie die Artikelnummer wie folgt aus:

```
output.println("Haeufigste Ausleihe: "+ artikelNummer);
```

3. Wie oft wurde ein Gerät im Durchschnitt ausgeliehen? Geben Sie das Ergebnis wie folgt aus:

```
output.println("Durchschnitt alle: "+ durchschnitt);
```

4. Wie oft wurde ein Gerät im Durchschnitt ausgeliehen, wenn nur Geräte berücksichtigt werden, die mindestens 3-mal ausgeliehen wurden? Wir definieren den Durchschnitt als 0, wenn kein Gerät mindestens 3-mal ausgeliehen wurde. Geben Sie das Ergebnis wie folgt aus:

```
output.println("Durchschnitt nachgefragte Geraete: "+ durchschnitt);
```

Jede der Informationen 1.–4. soll (in dieser Reihenfolge) auf einer separaten Zeile ausgegeben werden. Durchschnitte werden als ungerundete reelle Zahlen ausgegeben. Falls bei einer Information mehrere gleichwertige Ausgaben möglich sind, kann Ihr Programm *eine* beliebige der gleichwertigen Lösungen ausgeben.

Ihr Programm muss nur wohlgeformte, nicht-leere Eingabe-Dateien unterstützen. Ein Beispiel einer solchen Datei finden Sie im Projekt unter dem Namen “vermietungen.txt”. Exceptions im Zusammenhang mit Ein- und Ausgabe müssen nicht behandelt werden. Für die Beispiel-Datei sollte die Ausgabe wie folgt aussehen:

```
Laengste Zeit: 42
Haeufigste Ausleihe: 1
Durchschnitt alle: 0.006
Durchschnitt nachgefragte Geraete: 3.0
```

Testen Sie Ihr Programm ausgiebig—am besten mit JUnit—und pushen Sie die Lösung vor dem Abgabetermin.

Week 6

Dreiecksmatrix (2023 W6)

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 $Z \times S$ Dreiecksmatrix hat Z Zeilen X_0, X_1, \dots, 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 $D_{i,j}$ das $(j + 1)$ -te Element in der $(i + 1)$ -ten Zeile. $D_{0,0}$ ist das erste Element in der ersten Zeile [die immer genau 1 Element hat]. Abbildung 3 zeigt Beispiele von Dreiecksmatrizen. Beachten Sie, dass es möglich ist, dass zwei (aufeinanderfolgende) Zeilen die selbe Anzahl Elemente haben.

0,0	0,0	0,0
1,0 1,1	1,0 1,1	3,0 3,1 3,2 3,3
2,0 2,1 2,2 2,3	2,0 2,1 2,2	0,0
3,0 3,1 3,2 3,3 3,4	3,0 3,1 3,2 3,3	1,0
4,0 4,1 4,2 4,3 4,4 4,5 4,6		2,0 2,1

Abbildung 3: 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 $D_{i,j}$ zurück.

2. `void put(int i, int j, int value)` setzt das Element $D_{i,j}$ 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 $D_{i,j}$ den Wert von $t_{i,j}$, falls $t_{i,j}$ existiert. Falls $t_{i,j}$ nicht existiert, dann bleibt $D_{i,j}$ 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.

Prefixkonstruktion (2022 W6)

Gegeben seien zwei Strings s und t und ein Integer n mit $n \geq 0$. Schreiben Sie ein Programm, das zurückgibt, ob s eine Konkatenation von maximal n vielen Prefixen von t ist.

Beispiele:

- $s = \text{"abcbababc"} , t = \text{"abc"} , n = 4$: Das Programm sollte `true` zurückgeben, da "abc" und "ab" Prefixe von t sind und s eine Konkatenation von "abc", "ab", "abc" ist.
- $s = \text{"abcbcababc"} , t = \text{"abc"} , n = 4$: Das Programm sollte `false` zurückgeben, da "bc" kein Prefix von t ist.
- $s = \text{"abab"} , t = \text{"abac"} , n = 2$: Das Programm sollte `true` zurückgeben, da "ab" in ein Prefix von t ist und s eine Konkatenation von "ab", "ab" ist.

Implementieren Sie die Methode `isPrefixConstruction(String s, String t, int n)` in der Klasse `PrefixConstruction`. Die Methode hat drei Argumente: die beiden Strings s und t und der Integer n . Sie dürfen davon ausgehen, dass der Integer grösser oder gleich 0 ist. In der Datei "PrefixConstructionTest.java" finden Sie Tests. **Tipp:** Lösen Sie die Aufgabe rekursiv.

Split57 (2021 W6)

Gegeben ein Array x von Integern. Ein Paar von Arrays (a, b) ist eine *57-Partition* von x genau dann, wenn:

1. Jedes Element aus x entweder in a oder in b enthalten ist (und a und b keine weiteren Elemente enthalten).
2. Jedes Element aus x , das durch 5 teilbar ist, in a enthalten ist.

3. Jedes Element aus x , das durch 7, aber nicht durch 5 teilbar ist, in b enthalten ist.
4. Die Summe der Elemente aus a gleich der Summe der Elemente aus b ist.

Implementieren Sie die Methode `Split57.split57(int[] x)`. Die Methode nimmt als Argument einen Array x von Integern und gibt einen Array von Booleans zurück. Die Methode soll `null` zurückgeben, falls x keine 57-Partition hat. Falls x eine 57-Partition (a, b) hat, dann soll die Methode einen Boolean Array res zurückgeben, sodass res die gleiche Länge wie x hat und sodass $res[i]$ gleich `true` ist genau dann, wenn $x[i]$ in a enthalten ist. Ähnlich, $res[i]$ ist gleich `false` genau dann, wenn $x[i]$ in b enthalten ist. Beachten Sie, dass es für ein Argument mehr als eine 57-Partition geben kann und somit auch mehr als einen korrekten Rückgabewert geben kann. Zum Beispiel, wenn das Argument $[1, 1, 2]$ ist, dann können sowohl $[true, true, false]$ als auch $[false, false, true]$ zurückgegeben werden. Die Datei "Split57Test.java" enthält weitere Beispiele. Testen Sie Ihre Lösung ausgiebig —am besten mit JUnit—und pushen Sie die Lösung vor dem Abgabetermin.

Familienbeziehungen (2020 W6)

Mit dieser Bonus-Aufgabe sollen Sie zeigen, dass Sie einfache Familienbeziehungen aus einer anderen Sprache ins Standarddeutsche übersetzen können.

In einem fernen Land ¹ wird die Beziehung zwischen einer Person und ihren Vorfahren nach folgendem Muster ausgedrückt: Für eine Person P heisst die Mutter *mor* und der Vater *far*. Die Mutter von einem Vorfahren V hat dann an der Bezeichnung für V noch den Suffix *mor* und der Vater von V hat den Suffix *far*. Zum Beispiel der Vater (**far**) der Mutter (*mor*), also der Grossvater von P , ist die *morfar*. Die Grossmutter mütterlicherseits ist die *mormor*. Entsprechend werden die Vorfahren des Vaters bezeichnet. Der Grossvater väterlicherseits ist der *farfar*, die Grossmutter väterlicherseits ist die *farmor*. Dieses Muster wird auch für die Eltern der Grosseltern weitergeführt; der Vater der Grossmutter mütterlicherseits ist dann *mormorfar*, die Mutter des Grossvaters väterlicherseits ist dann *farfarmor*. Im Deutschen gibt es dafür nur zwei Begriffe, Urgrossmutter und Urgrossvater.

Ihre Aufgabe ist es, ein Programm zu schreiben, das in dieser Sprache ausgedrückte Familienbeziehung ins Standarddeutsche übersetzt. Dabei gehen Informationen verloren, wie es am Beispiel der Urgrossmutter gezeigt wurde. Diese Information sollen Sie in einem weiteren String zusammengefasst werden, nach folgendem Muster: Ein *w* steht für einen weiblichen Vorfahren, ein *m* für einen männlichen Vorfahren. Der erste Buchstabe von rechts drückt die Eltern aus, der zweite die Grosseltern, und so weiter.

Die folgende Tabelle zeigt einige Beispiele.

¹Diese Regeln entsprechen nicht ganz den Regeln einer wirklichen Sprache. Unsere Vereinfachungen erleichtern die Implementation.

Input	Output	Zusammenfassung
mor	Mutter	w
far	Vater	m
farmor	Grossmutter	wm
morfar	Grossvater	mw
farfar	Grossvater	mm
mormormor	Urgrossmutter	www
morfarmor	Urgrossmutter	wmw
mormormormor	Ururgrossmutter	www

Vervollständigen Sie die Methode `String[] toGerman(String name)` von der Klasse `Names`. Die Methode nimmt als Argument eine Familienbeziehung dieser Sprache und gibt einen Array mit zwei Strings zurück, wovon der erste String die Übersetzung von dem Argument ins Deutsche ist (Output in der obigen Tabelle) und der zweite String die Zusammenfassung ist (Zusammenfassung in der obigen Tabelle). Sie dürfen annehmen, dass das Argument nicht null und eine korrekte Bezeichnung dieser Sprache ist. Die Beschreibung einer Beziehung kann beliebig lang sein, ist aber immer endlich.

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 Klasse `NamesTest` bereits erstellt.

Schweizer Uhrzeit (2019 W6)

Laut Donald Knuth «*hat eine Person etwas erst richtig verstanden, nachdem sie es einem Computer beigebracht hat, d.h. es als Algorithmus ausgedrückt hat.*» In dieser Bonus-Aufgabe sollen Sie zeigen, dass Sie verstanden haben, wie in der Deutschschweiz die Uhrzeit ausgedrückt wird.

Vervollständigen Sie die Methode `toSwissGerman` in der Klasse `SwissTime`. Diese Methode nimmt als Parameter einen String mit dem Format `hh:mm`, wobei `hh` die Stunden und `mm` die Minuten sind, und soll einen String zurückgeben, der diese Uhrzeit auf Schweizerdeutsch enthält.

Beispiele:

```
00:00 -> 12i znacht
01:45 -> viertel vor 2 znacht
09:25 -> 5 vor halbi 10i am morge
12:01 -> 1 ab 12i am mittag
16:46 -> 14 vor 5i am namittag
21:51 -> 9 vor 10i am abig
22:37 -> 7 ab halbi 11i znacht
```

Wie Sie sehen, geht die Schweizer Zeit nur von 1 bis 12, dafür gibt es verschiedene Tageszeiten-Suffixe ("znacht", "am morge", usw.). Gegenüber dem `hh:mm`-Format wird die Stunde zudem um 1 erhöht, falls die Anzahl Minuten grösser oder gleich 25 ist ("5 vor halbi 10i"). Das Ausdrücken der Minuten selbst ist noch komplizierter: wenn es weniger als 25 sind, sagt man "ab", sonst grundsätzlich "vor"; allerdings zwischen Minute 25 und 39 sind sie "vor halbi" oder "ab halbi", und wenn es genau 15, 30 oder 45 sind, sagt man "viertel ab", "halbi" oder "viertel vor".

Im "test"-Ordner finden Sie eine grössere Menge von JUnit-Tests, welche das Format noch genauer spezifizieren. Diese (und weitere) werden für die Bewertung verwendet. Versuchen Sie

also, diese Tests zu lesen und zu verstehen, und passen Sie Ihre Lösung so lange an, bis sie alle davon besteht. Ihre Lösung muss nur korrekt formatierte Eingaben unterstützen.

PS: Falls die Spezifikation nicht Ihrem eigenen Dialekt entspricht, dürfen Sie gerne eine Kopie von `SwissTime` anfertigen und nach Ihrem Gusto gestalten. Für die Abgabe, d.h. in der Klasse `SwissTime`, müssen sich aber an die vorgegebene Spezifikation halten. :P

Talsoleh (2018 W6)



- a) In `"Tal.java"` finden Sie ein Programm, das eine Serie von Höhen (in Metern über Meer) aus der Datei `"gipfelhoehen.txt"` in das `int`-Array `hoehen` einliest. Dieses Array wird dann der Methode `findeGroessteVertiefung` übergeben, welche die *Vertiefung* der "vertieftesten" Talsoleh ermittelt und zurück gibt. Ihre Aufgabe ist es, diese Methode zu implementieren.

Eine *Talsoleh* ist ein Punkt oder eine Serie von gleich hohen Punkten. Die Serie muss dabei beidseitig durch höherliegende Punkte abgegrenzt sein. Analog dazu gibt es *Gipfel*, welche Punkte oder Serien sind, die beidseitig durch tieferliegende Punkte abgegrenzt sind. Der erste und der letzte Punkt im `hoehen`-Array gelten somit weder als *Gipfel* noch als *Talsoleh*.

Die *Vertiefung* einer Talsoleh wird anhand der benachbarten Gipfel gemessen. Die *Vertiefung* ist die Höhendifferenz zum niedrigsten benachbarten Gipfel. Die "vertiefteste" Talsoleh ist also nicht diejenige mit der niedrigsten Höhe in der Höhenserie, sondern diejenige, welche die grösste *Vertiefung* aufweist.

Um die Aufgabe zu vereinfachen, beginnt und endet die Höhenserie immer mit 0 und alle Höhen sind nicht-negativ. Falls keine Talsoleh in der Höhenserie vorkommt, oder `hoehen==null` gilt, soll 0 zurückgegeben werden.

In der Höhenserie `{0, 17, 17, 9, 8, 8, 11, 4, 8, 0}`, beispielsweise, gibt es zwei Talsolehen beginnend an den Positionen 4 und 7, und drei Gipfel beginnend an den Positionen 1, 6 und 8. Die Höhendifferenzen der ersten Talsoleh zu ihren beiden benachbarten Gipfeln betragen $17 - 8 = 9$ und $11 - 8 = 3$, somit ist die *Vertiefung* der Talsoleh 3. Die zweite Talsoleh hat zu ihren beiden benachbarten Gipfeln die Differenzen $11 - 4 = 7$ und $8 - 4 = 4$, und somit eine *Vertiefung* von 4. Damit sollte die Methode `findeGroessteVertiefung` für diese Höhenserie 4 zurückgeben.

- b) In der Datei `"TalTest.java"` finden Sie einen Test. Schreiben Sie weitere Tests um sicherzustellen, dass Ihre Implementierung richtig ist. Anschliessend committen und pushen Sie Ihre Lösung vor dem Abgabetermin.

Beachten Sie, dass wir nur die Methode `findeGroessteVertiefung` bewerten. Was Sie mit Ihren Tests und der `main`-Methode machen, wird nicht berücksichtigt. Bevor Sie Ihre Lösung committen und pushen, stellen Sie sicher, dass Sie die Methodensignatur

(`static int findeGroessteVertiefung(int[] hoehen)`) nicht verändert haben.

Week 7

Rechnungen (2023 W7)

In dieser Aufgabe sollen Sie einen Teil des Systems implementieren, das für den lokalen Stromversorger die Rechnungen erstellt.

Vervollständigen Sie die `process`-Methode in der Klasse `Bills`. Die Methode hat zwei Argumente: einen `Scanner`, von dem Sie den Inhalt der Eingabedatei lesen sollen, und einen `PrintStream`, in welchen Sie die unten beschriebenen Informationen schreiben.

Ihr Programm muss nur korrekt formatierte Eingabedateien unterstützen. Ein Beispiel einer solchen Datei finden Sie im Projekt unter dem Namen `"Data.txt"`. Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren.

Eine valide Eingabedatei enthält Zeilen, die entweder den Tarif, der angewendet werden soll, oder die Daten für den Stromverbrauch eines Kunden beschreiben. Der Verbrauch eines Kunden ist niemals grösser als 100000 Kilowattstunden.

Eine Tarifbeschreibung hat folgendes Format:

$$\text{Tarif_}n_l_1_p_1 \dots l_n_p_n$$

Folgendes gilt für die Parameter:

- Tarif (so geschrieben) ist ein Keyword, das angibt, dass die Zeile einen Tarif beschreibt.
- n ist eine positive ganze Zahl, welche die Anzahl der Intervalle angibt, für welche ein Strompreis festgelegt ist.
- Auf n folgt eine Folge von n Paaren von ganzen Zahlen $(l_1_p_1 \dots l_n_p_n)$. Die erste Zahl eines Paares gibt die Obergrenze des Intervalls an und die zweite den Preis für diesen Verbrauch; für ein i , so dass $1 \leq i \leq n$, ist l_i also der Verbrauch (in Kilowattstunden), bis zu welchem der Strompreis p_i (in Rappen pro Kilowattstunde) zur Anwendung kommt ($l_i > 0$ und $p_i \geq 0$). Die Paare sind jeweils mit einem Whitespace voneinander getrennt (und l_i und p_i jeweils voneinander auch).

Hier sind einige Beispiele für Tarifbeschreibungen:

1. Tarif 1 100000 30

Es gibt ein Intervall und für jede Kilowattstunde müssen 30 Rappen bezahlt werden.

2. Tarif 2 1000 10 100000 30

Es gibt zwei Intervalle. Die ersten 1000 Kilowattstunden kosten 10 Rappen pro Kilowattstunde. Der Rest kostet 30 Rappen pro Kilowattstunde.

3. Tarif 3 100 40 1000 10 100000 30

Es gibt drei Intervalle. Die ersten 100 Kilowattstunden kosten 40 Rappen pro Kilowattstunde. Die nächsten 1000 Kilowattstunden kosten 10 Rappen pro Kilowattstunde. Der Rest kostet 30 Rappen pro Kilowattstunde.

Wenn ein Kunde im Jahr 2000 Kilowattstunden verbraucht, so beträgt die Rechnung für das erste Beispiel 600 Franken, im zweiten Beispiel 400 Franken und 410 Franken im dritten.

Die Beschreibung des Stromverbrauchs eines Kunden hat folgendes Format:

$$ID_v_{q_1}_v_{q_2}_v_{q_3}_v_{q_4}$$

Hierbei gilt für die Parameter:

- ID ist eine positive ganze Zahl.
- v_{q_1} ist eine ganze Zahl, die den Verbrauch im ersten Quartal in Kilowattstunden angibt ($v_{q_1} \geq 0$).
- v_{q_2} ist eine ganze Zahl, die den Verbrauch im zweiten Quartal in Kilowattstunden angibt ($v_{q_2} \geq 0$).
- v_{q_3} ist eine ganze Zahl, die den Verbrauch im dritten Quartal in Kilowattstunden angibt ($v_{q_3} \geq 0$).
- v_{q_4} ist eine ganze Zahl, die den Verbrauch im vierten Quartal in Kilowattstunden angibt ($v_{q_4} \geq 0$).

Hier ist ein Beispiel für eine Verbrauchbeschreibung:

115 0 0 0 2000

Der Kunde mit ID 115 hat nur im vierten Quartal Strom verbraucht. Da waren es 2000 Kilowattstunden.

Ein einmal gelesener Tarif wird für alle Kunden angewendet, die nach dieser Tarifinformation in der Eingabedatei erscheinen. Wenn ein neuer Tarif erscheint, dann gilt der danach für die weiteren Kunden bis auf Weiteres. Sie können davon ausgehen, dass eine Kunden-ID nur einmal in der Eingabedatei vorkommen kann und dass die erste Zeile der Eingabedatei eine Tarifbeschreibung ist.

Die Methode `process` soll die Eingabedatei verarbeiten und für jeden Kunden eine Zeile

$$ID_b$$

in den der Methode in `output` übergebenen `PrintStream` schreiben. ID ist die ID des Kunden (`int`) und b ist eine **ganze** Zahl, die die jeweilige Rechnung für den Jahresverbrauch **in Franken** angibt. (Zuerst muss der Jahresverbrauch berechnet werden, dann kann der entsprechende Tarif angewendet werden.) Berechnen Sie den Rechnungsbetrag und runden Sie das Resultat anschliessend (vor der Ausgabe, aber nach den Berechnungen) auf die nächste ganze Zahl. Sie können hierfür die Methode `Math.round(double a)` verwenden. Die Ausgabe darf keine weiteren Zeichen enthalten. Sie können den Betrag so ausgeben, wie er von der `println`-Anweisung herausgegeben wird, d.h. Sie brauchen das Ergebnis nicht zu formatieren.

In der Datei `"BillsTest.java"` finden Sie einen einfachen Test, um das Format Ihres Outputs zu testen.

Tip: Sie können die Aufgabe ohne weitere Vorgaben implementieren. Wir empfehlen, dass Sie sich überlegen, was sinnvolle Klassen sein könnten und was für Teilaufgaben (die dann als Methode implementiert werden können) zweckmässig sind.

Roulette (2022 W7)

In dieser Aufgabe implementieren Sie ein vereinfachtes, an Roulette angelehntes Glücksspiel. Das Spiel besteht aus Runden. Pro Runde wird eine von 37 Zahlen ausgewählt (von 0 bis 36) und dann werden den teilnehmenden Spielern entsprechende Gewinne ausgezahlt. Spieler müssen sich bei einem Spiel registrieren, um an den folgenden Runden teilnehmen zu können. Registrierte Spieler können eine Wette abgeben, um an der nächsten Runde eines Spiels teilzunehmen. Eine Wette besteht aus einem Einsatz, welchen der Spieler bezahlt, um teilzunehmen, und einem Set von Zahlen, auf das gesetzt wird. Dabei gibt es drei Optionen: auf alle gerade Zahlen zu setzen (ohne 0), auf alle ungeraden Zahlen zu setzen, und auf eine einzige Zahl zu setzen. Eine Wette ist erfolgreich, falls die Zahl, welche in der nächsten Runde ausgewählt wird, im Set der gesetzten Zahlen der Wette enthalten ist. Bei einer erfolgreichen Wette erhält der Spieler, welche die Wette abgegeben hat, einen Gewinn (bei einer nicht erfolgreichen Wette gibt es keinen Gewinn). Für eine erfolgreiche Wette gibt es den doppelten Einsatz zurück, falls auf alle geraden oder ungeraden Zahlen gesetzt wurde, und es gibt den 36-fachen Einsatz zurück, falls auf eine einzige Zahl gesetzt wurde. Eine Wette ist immer nur für die folgende Runde gültig. Spieler haben einen Kontostand, der verwendet wird, um die Einsätze von Wetten zu bezahlen und an den Gewinne zurückgezahlt werden. Jeder Spieler kann für jede Runde immer nur eine geltende Wette haben. Falls ein Spieler mehrer Wetten für eine Runde setzt, dann ist immer nur die letzte Wette gültig.

Die Klasse `RPlayer` repräsentiert einen Spieler. Die Klasse hat einen Konstruktor `RPlayer(int start)`, wobei das Argument `start` den anfänglichen Kontostand angibt. Zusätzlich hat die Klasse die folgenden Methoden:

- `RPlayer.getBalance()` gibt den aktuellen Kontostand vom Spieler zurück.
- `RPlayer.setNumber(int number, int bet)` gibt eine Wette für die einzige Zahl `number` mit Einsatz `bet` ab. Sie dürfen annehmen, dass $0 \leq \text{number} \leq 36$ und $0 \leq \text{bet}$ gilt. Die Wette wird nicht gesetzt, falls der Kontostand des Spielers kleiner als `bet` ist. In dem Fall gibt die Methode `false` zurück. Falls der Kontostand ausreicht, wird der Einsatz vom Kontostand des Spielers abgezogen, die Wette gesetzt, und `true` zurückgegeben.
- `RPlayer.setOdd(int bet)` gibt eine Wette für alle ungeraden Zahlen mit Einsatz `bet` ab. Sie dürfen annehmen, dass $0 \leq \text{bet}$ gilt. Die Wette wird nicht gesetzt, falls der Kontostand des Spielers kleiner als `bet` ist. In dem Fall gibt die Methode `false` zurück. Falls der Kontostand ausreicht, wird der Einsatz vom Kontostand des Spielers abgezogen, die Wette gesetzt, und `true` zurückgegeben.
- `RPlayer.setEven(int bet)` gibt eine Wette für alle geraden Zahlen (ohne 0) mit Einsatz `bet` ab. Sie dürfen annehmen, dass $0 \leq \text{bet}$ gilt. Die Wette wird nicht gesetzt, falls der Kontostand des Spielers kleiner als `bet` ist. In dem Fall gibt die Methode `false` zurück. Falls der Kontostand ausreicht, wird der Einsatz vom Kontostand des Spielers abgezogen, die Wette gesetzt, und `true` zurückgegeben.

Die Klasse `Roulette` repräsentiert ein Spiel und hat folgende Methoden:

- `Roulette.register(RPlayer player)` registert den Spieler `player` an dem Spiel. Sie dürfen annehmen, dass `player` bei keinem anderen Spiel registriert und dass maximal 10 Spieler bei einem Spiel registriert werden.

- `Roulette.play()` wählt eine zufällige Zahl aus (zwischen 0 bis einschliesslich 36) und führt damit wie oben beschrieben eine Runde durch. Die Methode gibt die ausgewählte Zahl zurück.
- `Roulette.force(int number)` verwendet `number` als ausgewählte Zahl und führt damit wie oben beschrieben eine Runde durch. Sie dürfen annehmen, dass $0 \leq \text{number} \leq 36$ gilt.

Implementieren Sie alle aufgeführten Methoden und Konstruktoren. Die Datei "RouletteTest.java" enthält eine kommentierte Durchführung des Spiels. Wir empfehlen, dass Sie sich diese Datei ansehen, bevor Sie die Aufgabe lösen. Beachten Sie, dass die Methode `Roulette.force(int number)` wichtig ist, damit wir Ihre Lösung testen können. Es kann sein, dass Sie keine Punkte erhalten, falls `Roulette.force(int number)` nicht implementiert ist. Daher sollten Sie `Roulette.force(int number)` implementieren, auch wenn Ihre Lösung nicht alle Arten von Wetten unterstützt.

Grid (2021 W7)

Die Klasse `Node` repräsentiert einen Knoten in einem gerichteten Graphen, wobei es für jeden Knoten g höchstens zwei gerichtete Kanten von g zu anderen Knoten f, h geben kann (f, g und h können gleich sein). Wir unterscheiden dabei zwischen der rechten und unteren Kante (und damit dem rechten und unteren Knoten). Die Methode `Node.getRight()` (bzw. `Node.getBottom()`) gibt den rechten Knoten (bzw. unteren Knoten) zurück (als `Node`-Objekt). Wenn der rechte Knoten von n_0 nicht existiert, dann gibt `Node.getRight()` `null` zurück (analog für den unteren Knoten). Die Methode `Node.setRight(Node r)` (bzw. `Node.setBottom(Node b)`) setzt den rechten (bzw. unteren Knoten).

In dieser Aufgabe geht es um `Node`-Objekte, welche ein $N \times M$ Gitter G mit N Zeilen und M Spalten modellieren ($N, M > 0$). Die $N \cdot M$ unterschiedlichen `Node`-Objekte $\{n_{i,j} | 0 \leq i < N, 0 \leq j < M\}$ modellieren die Knoten $G_{i,j}$ auf Position (i, j) im Gitter genau dann, wenn folgendes gilt (wobei $0 \leq i < N, 0 \leq j < M$):

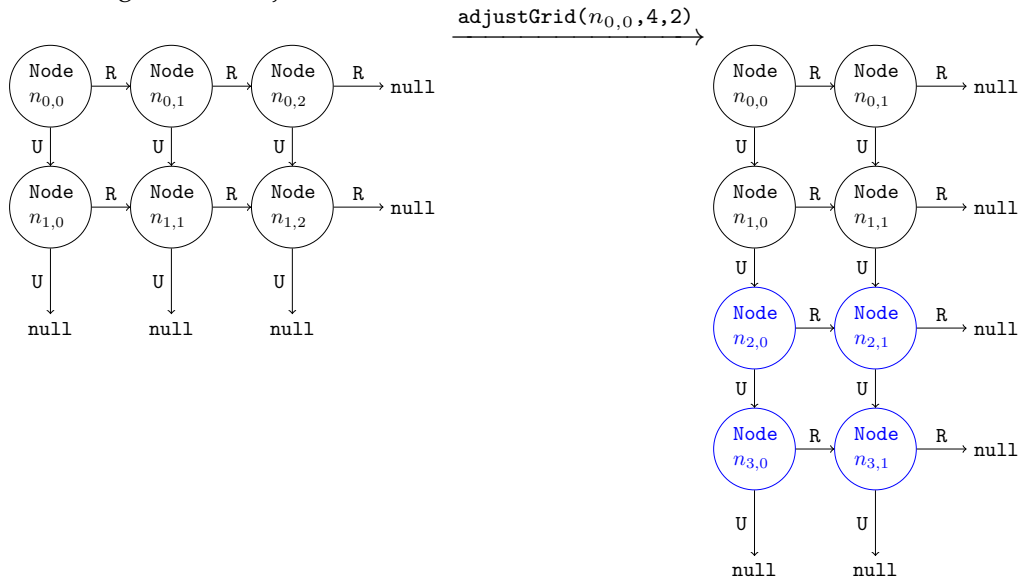
- Der rechte Knoten (bzw. untere Knoten) von $n_{i,j}$ existiert genau dann, wenn $j < M - 1$ (bzw. $i < N - 1$).
- Der rechte Knoten von $n_{i,j}$ ist $n_{i,j+1}$ (wenn der rechte Knoten existiert) und der untere Knoten von $n_{i,j}$ ist $n_{i+1,j}$ (wenn der untere Knoten existiert).

Implementieren Sie die Methode `Grid.adjustGrid(Node origin, int A, int B)`. Der erste Parameter ist ein `Node`-Objekt `origin`, welches den Ursprungsknoten $G_{0,0}$ auf Position $(0,0)$ eines $N \times M$ Gitters G ($N, M > 0$) modelliert (`origin` ist nie `null`). Nach dem Aufruf von `adjustGrid` muss `origin` den Ursprungsknoten $H_{0,0}$ eines $A \times B$ Gitters H modellieren (A und B sind die Parameter von `adjustGrid` und es gilt immer $A, B > 0$). Für die Knoten von H muss zusätzlich gelten:

- Für alle i, j mit $0 \leq i < \min(A, N)$ und $0 \leq j < \min(B, M)$ muss der Knoten $H_{i,j}$ durch das bestehende `Node`-Objekt $n_{i,j}$, welches vor dem Aufruf von `adjustGrid` $G_{i,j}$ modelliert, modelliert werden (mit einem potenziell anderen rechten bzw. unteren Knoten als vor dem Aufruf von `adjustGrid`).

- Für Knoten in H , die nicht aus G übernommen werden, gelten nur die oben aufgeführten Bedingungen für die Modellierung eines $A \times B$ Gitters.

Im folgenden Beispiel sehen Sie den Effekt des Aufrufs `adjustGrid($n_{0,0}$, 4, 2)` für ein 2×3 Gitter. Das Bild links zeigt die Verlinkung vor dem Aufruf und das Bild rechts zeigt die Verlinkung nach dem Aufruf. Die **blauen** Node-Objekte können neu erstellt werden, oder Sie können nicht mehr benötigte Node-Objekte verwenden. R und U verweisen auf den rechten bzw. unteren Knoten.



Beachten Sie dass in diesem Beispiel $n_{1,0}$ sowie vor als auch nach dem Aufruf den Knoten auf Position (1,0) modelliert. Vor dem Aufruf ist der Verweis auf den unteren Knoten von $n_{1,0}$ null, nach dem Aufruf ist der untere Knoten $n_{2,0}$.

Wir geben zwei Testdateien zur Verfügung. "GridTest.java" enthält Tests, welche wir an einer Prüfung geben würden. "GridTestExam.java" enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie ihre Lösung zuerst ausgiebig mit "GridTest.java" (am besten fügen Sie selber neue Tests hinzu) und dann können Sie "GridTestExam.java" verwenden, um zu sehen wie ihre Lösung an einer Prüfung abgeschnitten hätte.

Subtraktion von Listen (2020 W7)

Auf dem letzten Übungsblatt haben Sie eine Linked List für Integer implementiert. In dieser Aufgabe arbeiten wir an einer ähnlichen Liste `SpecialIntList`. Der Unterschied ist, dass ein Node von `SpecialIntList` kein Feld mit dem Integer Wert hat, sondern stattdessen ein Feld mit einer `IntBox`, welche dann das Feld mit dem Integer Wert hat. Diese zusätzliche Indirektion erlaubt, dass mehrere Listen die gleichen Boxen verwenden können. Fügen Sie der Klasse `SpecialIntList` eine weitere Methode `SpecialIntList subtract(SpecialIntList list)` hinzu, welche eine `SpecialIntList` als Parameter nimmt und eine `SpecialIntList` zurückgibt. Die zurückgegebene Liste enthält alle Boxen der Liste, auf welcher die Methode aufgerufen wurde, ausser der Wert der Box ist in der Liste, welche als Argument übergeben wird. Das heisst, falls eine Box einen Wert enthält, der auch in der Liste des Funktionsargumentes auftritt, dann ist diese Box nicht in der zurückgegebenen Liste enthalten. Die Boxen, der zurückgegebenen Liste sollen dabei die gleiche

Reihenfolge der ursprünglichen Liste beibehalten. Die Liste, auf welcher die Methode aufgerufen wird, und die Liste, welche als Argument übergeben wird, dürfen dabei nicht verändert werden. Beachten Sie, dass die Boxen aus der zurückgegebenen Liste wirklich die gleichen Boxen sind wie aus der Liste, auf welcher die Methode aufgerufen wurde. Das bedeutet, dass Änderungen der einen Liste auch die andere Liste betreffen können, wie wir an folgendem Beispiel illustrieren:

```
SpecialLinkedList listA = ... // Liste mit den Werten 3, 7, 5, 5, 2
SpecialLinkedList listB = ... // Liste mit den Werten 5, 3, 8

// Resultiert in einer Liste mit den Werten 7, 2 (diese Reihenfolge!)
SpecialLinkedList result = listA.subtract(listB);

// Updated den Wert 2 zu dem Wert 8
result.set(1, 8);

// Weil result und listA die gleichen Nodes haben,
// ist auch listA von dem Update des Wertes betroffen.
assert(listA.get(4) == 8);

// Das Gleiche gilt auch umgekehrt.
listA.set(4, 52);
assert(result.get(1) == 52);
```

In der Datei "SpecialIntListTest.java" finden Sie den obigen Test. Wie immer, Sie können weitere Methoden und Felder hinzufügen. Ändern Sie aber nicht die anderen bestehenden Methoden und Felder. Bei der Bewertung können wir Tests hinzufügen, um zu prüfen, dass die anderen Methoden sich verhalten wie erwartet.

Split (2019 W7)

Auf dem letzten Übungsblatt haben Sie eine Linked List für Integer implementiert. In dieser Aufgabe fügen Sie einer ähnlichen Linked List eine weitere Methode `split` hinzu, welche einen Integer als Parameter nimmt und eine Linked List zurückgibt. Die Methode entfernt aus der Liste auf der sie aufgerufen wird alle Elemente, welche grösser als das Argument sind, und fügt diese entfernten Elemente in der gleichen Reihenfolge der zurückgegebenen Liste hinzu.

Wir verwenden dafür `SpecialIntLinkedList` und `SpecialIntNode` anstatt `IntLinkedList` und `IntNode`. Der Unterschied ist, dass `SpecialIntNode` neben einem `next` Feld auch ein Feld `oldNext` hat. Dieses Feld soll nach einem Aufruf von `split` den `SpecialIntNode` enthalten, welcher dem Nachfolger aus der Ursprünglichen Liste entspricht.

Abbildung 4 zeigt ein Beispiel: Wir beginnen mit einer Liste `This`, welche die Werte 3, 7, 2, 4, und 5 enthält. Die grünen Pfeile repräsentieren das `next` Feld, wobei kein Pfeil gezeichnet ist, wenn das Feld `null` enthält. Danach wird `split(4)` auf `This` aufgerufen, wobei die Liste `Result` zurückgegeben wird. `This` enthält danach die Werte kleiner oder gleich 4, nämlich 3, 2, und 4, während `Result` die Werte grösser als 4 enthält, nämlich 7 und 5. Zusätzlich zeigt das Feld `oldNext`, welches mit roten Pfeilen repräsentiert wird, auf den entsprechenden Nachfolger aus der ursprünglichen Liste, so dass man erneut 3, 7, 2, 4, und 5 enthält, wenn man den roten Pfeilen folgt.

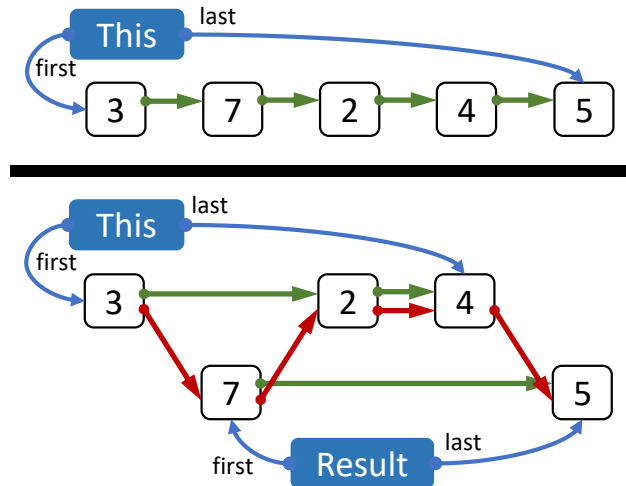


Abbildung 4: Liste vor und nach dem Aufruf von `This.split(4)`. Die Liste aus welcher `split` aufgerufen wird ist `This` und die zurückgegebene Liste ist `Result`. Grüne Pfeile repräsentieren das `next` Feld und rote Pfeile repräsentieren das `oldNext` Feld. Nicht gezeigte Pfeile entsprechen null.

Die `SpecialIntNode` Instanzen müssen nicht den gleichen Instanzen aus der Ursprungsliste entsprechen, das heißt, dass neue `SpecialIntNode` Instanzen erzeugt werden dürfen.

Vervollständigen Sie die Methode `split()` in der Klasse `SpecialLinkedList`. Die Methode soll, wie oben definiert, die Liste splitten. In der Datei `"Split.java"` befindet sich ein einfacher Klient um die Methode auszuführen und in der Datei `"SplitTest.java"` finden Sie ein paar einfache Tests.

Bitte ändern sie die Klassen `SpecialIntLinkedList` und `SpecialIntNode` nur wo vorgesehen ab.

Verzahnungen (2018 W7)

Gegeben seien zwei Strings `s` und `t`, die aus unterschiedlichen Buchstaben bestehen. (Das heißt, dass `s` und `t` keine Buchstaben gemeinsam haben und kein Buchstabe doppelt oder häufiger auftritt.) Schreiben Sie ein Programm, das alle möglichen Verzahnungen von `s` und `t` generiert.

Beispiel: `s = "12"`, `t = "ab"`. Dann lauten alle Verzahnungen: `12ab`, `1a2b`, `1ab2`, `a12b`, `a1b2`, und `ab12`.

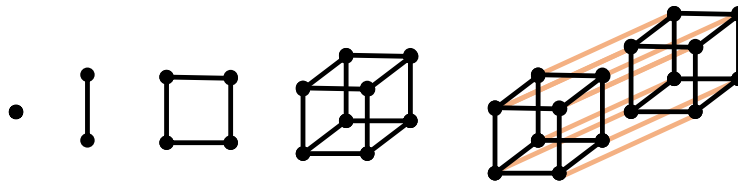
Vervollständigen Sie die Methode `verzahnungen()` in der Klasse `Verzahnungen`. Die Methode hat drei Argumente: die beiden Eingabestrings, `s` und `t`, und zusätzlich einen `PrintStream`, auf dem alle Verzahnungen ausgegeben werden sollen. Sie dürfen davon ausgehen, dass alle Argumente nie null sind. Auf dem `PrintStream` soll für jede Verzahnung genau eine Zeile, welche nur die Verzahnung enthält, geprintet werden. Die Reihenfolge der Verzahnungen ist nicht relevant. In der Datei `"VerzahnungenTest.java"` finden Sie einen Test, der prüft, ob Ihr Output-Format korrekt ist. **Tipp:** Lösen Sie zuerst Aufgabe 1.

Week 8

Graphgenerierung (2023 W8)

In dieser Aufgabe implementieren Sie ein Program, das bestimmte Graphen generiert. Die Klasse `Node` repräsentiert einen Knoten eines Graphen. Die Methode `Node.getNeighbors()` gibt alle Nachbarn eines Knotens zurück. Ein Knoten B ist ein Nachbar von A , falls es im Graphen eine gerichtete Kante von A zu B gibt.

Ein Hyperwürfel mit n -Dimensionen ist ein Graph mit 2^n Knoten. Um den Graphen zu erklären, weisen wir jedem Knoten ein einzigartiges Label zu. Als Label verwenden wir Sequenzen aus 0, 1 der Länge n , e.g. für $n = 3$ mit 8 Knoten sind die Labels 000, 001, 010, 011, 100, 101, 110, 111. Für ein Label x ist x_i das i -te Element der Sequenz (beginnend bei 0). Ein Knoten A mit Label x hat eine Kante zu einem Knoten B mit Label y genau dann wenn, die Labels x und y sich nur an einer Stelle unterscheiden. Das heisst, $\sum_{i=0}^{n-1} |x_i - y_i| = 1$ gilt. Das folgende Bild zeigt Hyperwürfel für n gleich 0 (links) bis 4 (rechts). Die Punkte stellen Knoten dar, welche durch Kanten verbunden sind. Bei dem Graphen für $n = 4$ benutzen wir für die leichtere Verständlichkeit zwei verschiedene Farben für die Kanten. Die Farben haben keine weitere Bedeutung.



Implementieren Sie die Methode `Graphs.cube(int n)`. Die Methode gibt einen Knoten zurück von einem Hyperwürfel mit n Dimensionen. Sie dürfen annehmen, dass n nicht negativ ist. Die Methode darf einen beliebigen Knoten des Hyperwürfels zurückgeben. **Tipp:** Vielleicht implementieren Sie die Methode rekursiv.

Die Datei `GraphsTest.java` enthält Tests. Wir geben als Unterstützung für den Entwicklungsprozess zusätzlich eine Methode `Graphs.printGraph(Node node)`, welche den Graphen von `node` als String über die Konsole ausgibt.

Executable Graph (2022 W8)

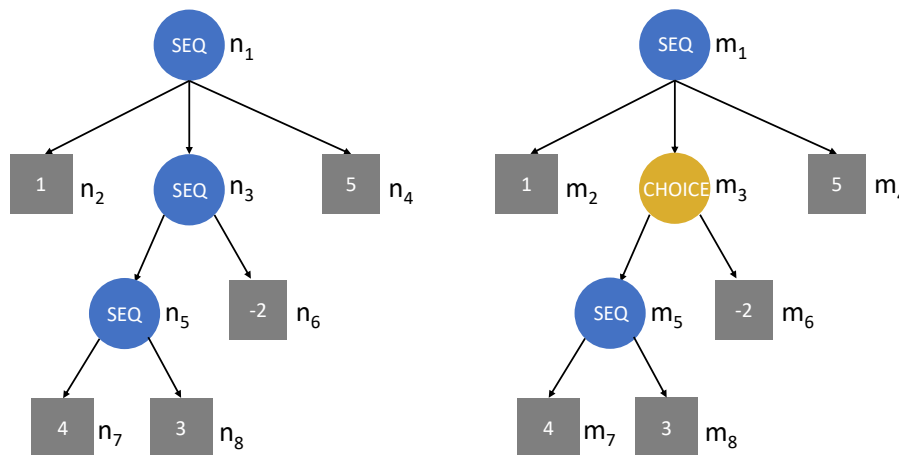
In dieser Aufgabe verwenden wir gerichtete azyklische Graphen, um Programme zu repräsentieren. Der Programmzustand ist dabei immer durch ein Tupel $(sum, counter)$ gegeben, wobei sum und $counter$ ganze Zahlen sind. Programmmzustände werden durch `ProgramState`-Objekte modelliert, wobei `ProgramState.getSum()` (bzw. `ProgramState.getCounter()`) dem ersten Element (bzw. dem zweiten Element) des Tupels entspricht.

Eine Ausführung des Programms manipuliert den Programmzustand und das Resultat eines Programms ist gegeben durch den erreichten Programmzustand, nachdem alle Operationen im Programm ausgeführt wurden. Programme können nichtdeterministisch sein: Das heisst, für ein einzelnes Programm kann es für den gleichen Startzustand mehrere Programmausführungen geben, welche zu unterschiedlichen Resultaten führen.

Knoten in Graphen werden durch Node-Objekte modelliert. `Node.getSubnodes()` gibt die Kinderknoten als ein Array zurück (m ist genau dann ein Kinderknoten von n , wenn es eine ausgehende gerichtete Kante von n zu m gibt). Wir unterscheiden drei Arten von Knoten, wobei die Methode `Node.getType()` die Knotenart als String zurückgibt. Um ein Programm, welches durch den Knoten n repräsentiert wird, auszuführen, muss man den "Knoten n ausführen". Wir beschreiben nun die drei Knotenarten und jeweils die Ausführung der Knoten:

1. **Additionsknoten** (`Node.getType()` ist "ADD"): Solche Knoten besitzen einen Additionswert a gegeben durch `Node.getValue()` (eine ganze Zahl) und bei der Ausführung dieses Knotens wird der Programmzustand von $(sum, counter)$ zu $(sum + a, counter + 1)$ aktualisiert. Die Kinderknoten von solchen Knoten werden bei der Ausführung ignoriert.
2. **Sequenzknoten** (`Node.getType()` ist "SEQ"): Bei der Ausführung eines Sequenzknoten n werden die Kinderknoten von n nacheinander ausgeführt. Die Reihenfolge in welcher die Kinderknoten ausgeführt werden spielt keine Rolle, da der erreichte Programmzustand für jede Reihenfolge gleich ist. `Node.getValue()` ist irrelevant.
3. **Auswahlknoten** (`Node.getType()` ist "CHOICE"): Bei der Ausführung eines Auswahlknoten n wird ein beliebiger Kinderknoten von n ausgewählt und ausgeführt. `Node.getValue()` ist irrelevant. Diese Knoten führen zu Nichtdeterminismus.

Sie dürfen davon ausgehen, dass Sequenz- und Auswahlknoten immer mindestens einen Kinderknoten haben, und dass es zwischen zwei Knoten immer höchstens einen Pfad gibt. Die folgende Abbildung zeigt zwei Beispielgraphen, wobei Knoten mit der Beschriftung "SEQ" (bzw. "CHOICE") Sequenzknoten (bzw. Auswahlknoten) entsprechen und die Zahlen in Additionsknoten den Additionswerten entsprechen.



Beim linken Graphen in der Abbildung gibt es immer nur eine mögliche Ausführung von n_1 pro Startzustand. Für den Startzustand $(1, 2)$ ist das Resultat gegeben durch $(1 + 1 + 4 + 3 - 2 + 5, 2 + 5) = (12, 7)$. Beim rechten Graphen gibt es zwei mögliche Ausführungen von m_1 . Beim Auswahlknoten m_3 wird entweder m_5 oder m_6 ausgeführt (da m_5 und m_6 die Kinderknoten von m_3 sind). Die beiden möglichen Resultate für den Startzustand $(0, 0)$ sind $(0 + 1 + 4 + 3 + 5, 0 + 4) = (13, 4)$ (wenn m_5 gewählt wird) und $(0 + 1 - 2 + 5, 0 + 3) = (4, 3)$ (wenn m_6 gewählt wird).

Implementieren Sie `GraphExecution.allResults(Node n, ProgramState initState)`, welche für den Startzustand `initState` alle möglichen Resultate für das Programm repräsentiert durch `n` zurückgibt. Die Resultate sollten als eine Liste von `ProgramState`-Objekten zurückgegeben werden (repräsentiert durch die Klasse `LinkedProgramStateList`). Die Reihenfolge der zurückgegebenen Liste spielt keine Rolle. Wenn das gleiche Resultat durch genau k verschiedene Ausführungen generiert werden kann, dann muss das Resultat k Mal in der zurückgegebenen Liste vorkommen. Zwei Ausführungen sind unterschiedlich, wenn es mindestens einen Knoten gibt, der in einer aber nicht in der anderen Ausführung ausgeführt wird.

Wir geben zwei Testdateien zur Verfügung. „`GraphExecutionTest.java`“ enthält Tests, welche wir an einer Prüfung geben würden. „`GradingGraphExecutionTest.java`“ enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie ihre Lösung zuerst ausgiebig mit „`GraphExecutionTest.java`“ (am besten fügen Sie selber neue Tests hinzu) und dann können Sie „`GradingGraphExecutionTest.java`“ verwenden, um zu sehen wie ihre Lösung an einer Prüfung abgeschnitten hätte.

Labyrinth (2021 W8)

Ein Labyrinth besteht aus einer Menge von Räumen, welche durch die Klasse `Room` dargestellt werden. Die Klasse hat zwei Attribute: Der Integer `age` (grösser gleich 0) beschreibt das Alter des Raums und der Array `doorsTo` (nie `null`) beschreibt die Türen von diesem Raum zu anderen Räumen. Alle Türen sind Falltüren, d.h. sie funktionieren nur in eine Richtung. Ein Raum ist ein Ausgang aus dem Labyrinth, wenn keine Türen von dem Raum wegführen, das heisst, wenn `doorsTo` eine Länge von 0 hat.

Für alle Aufgaben werden Sie in einen zufälligen Raum geworfen, welcher als Argument gegeben wird (garantiert nicht `null`) und von welchem aus Sie die Aufgabe lösen müssen. Sie dürfen für alle Aufgaben annehmen, dass es im Labyrinth keinen Zyklus gibt. Das heisst, dass man einen Raum, welchen man durch eine Tür verlassen hat, nie wieder erreichen kann indem man weiteren Türen folgt. Eine Sequenz von N Räumen r_1, \dots, r_N ist ein *Lösungspfad* für einen Raum `room` genau dann wenn: (1) Der erste Raum r_1 ist der Raum `room`, (2) der letzte Raum r_N ist ein Ausgang, und (3) jeder Raum r_i mit $1 \leq i < N$ hat eine Tür zum nächsten Raum in der Sequenz r_{i+1} .

a) Implementieren Sie die Methode `Labyrinth.task1(Room room)`. Die Methode soll `true` zurückgeben genau dann, wenn es einen Lösungspfad r_1, \dots, r_N für `room` gibt, sodass:

- Für jede Teilsequenz r_1, \dots, r_i mit $1 \leq i \leq N$ gilt, dass die Summe der Alter der Räume r_1, \dots, r_i nicht durch 3 teilbar ist.

b) Implementieren Sie die Methode `Labyrinth.task2(Room room)`. Die Methode soll `true` zurückgeben genau dann, wenn es zwei Lösungspfade r_1, \dots, r_N und s_1, \dots, s_N für `room` gibt, sodass:

- Die Räume r_i und s_i haben das gleiche Alter für jedes i mit $1 \leq i \leq N$.
- Für mindestens ein i mit $1 \leq i \leq N$ gilt, dass r_i und s_i unterschiedlich sind (verschiedene Referenzen).

Sie dürfen Methoden und Felder der Klasse `Room` hinzufügen. Tests finden Sie in der Datei `"LabyrinthTest.java"`. **Tipp:** Lösen Sie die Aufgaben rekursiv. Für keine der Aufgaben müssen Sie alle Pfade generieren und dann erst prüfen, dass die Eigenschaften gelten. Manche der Tests enthalten Labyrinth mit einer extrem grossen Anzahl an Pfaden aber leichten Lösungen.

Mindestanzahl an Teilfolgen (2020 W8)

Gegeben seien zwei Strings `s` und `t` und ein Integer `n` mit `t ≠ ""` und `n ≥ 0`. Schreiben Sie ein Programm, das zurückgibt, ob `t` in `s` mindestens `n` mal als Teilfolge vorkommt.

Beispiele:

- `s = "abab"`, `t = "ab"`, `n = 3`. Das Programm sollte `true` zurückgeben, da `"ab"` in `s` dreimal als Teilfolge vorkommt (Das erste `"a"` einmal mit dem ersten und einmal mit dem zweiten `"b"` und das zweite `"a"` einmal mit dem zweiten `"b"`).
- `s = "abab"`, `t = "ab"`, `n = 4`. Das Programm sollte `false` zurückgeben, da `"ab"` in `s` nur dreimal als Teilfolge vorkommt
- `s = "abab"`, `t = "ab"`, `n = 1`. Das Programm sollte `true` zurückgeben, da `"ab"` in `s` mindestens einmal als Teilfolge vorkommt

Vervollständigen Sie die Methode `boolean subsequence(String s, String t, int n)` in der Klasse `AtLeastSubsequence`. Die Methode hat drei Argumente: die beiden Strings `s` und `t` und der Integer `n`. Sie dürfen davon ausgehen, dass die Strings nie `null` sind, dass `t` mindestens ein Zeichen enthält, und dass der Integer grösser oder gleich 0 ist. In der Datei `"AtLeastSubsequenceTest.java"` finden Sie Tests. **Tipp:** Lösen Sie die Aufgabe rekursiv.

Enthalten mit Abstand (2019 W8)

Gegeben seien zwei Strings `s` und `t` und ein Integer `k` mit `k ≥ 0`. Schreiben Sie ein Programm, das zurückgibt ob die Zeichen aus `t` in beliebiger Reihenfolge als Subsequenz in `s` vorkommen, in welcher die in der Subsequenz aufeinanderfolgende Zeichen *maximal* einen Abstand von `k` in `s` haben. Der Abstand zwischen zwei Zeichen `a` und `b` in einer Sequenz ist die Anzahl Zeichen zwischen `a` und `b`. Zum Beispiel in `"a12345b"` haben das `a` und das `b` einen Abstand von 5.

Beispiele:

- `s = "abbbbc"`, `t = "cab"`, `k = 1`. Das Programm sollte `true` zurückgeben, da `"abc"` in `s` als Subsequenz mit Abstand maximal 1 vorkommt (Die Subsequenz in `s` ist das erste, das dritte, und das letzte Zeichen).
- `s = "abbbbc"`, `t = "cab"`, `k = 1`. Das Programm sollte `false` zurückgeben, da entweder der Abstand zwischen `"a"` und `"b"` oder der Abstand zwischen `"b"` und `"c"` immer grösser ist als 1.
- `s = "abc"`, `t = "cbab"`, `k = 1`. Das Programm sollte `false` zurückgeben, da es keine Subsequenz in `s` gibt, welche zwei `"b"` enthält.

Vervollständigen Sie die Methode `enthalten()` in der Klasse `EnthaltenMitAbstand`. Die Methode hat drei Argumente: die beiden Strings `s` und `t` und der Integer `k`. Sie dürfen davon ausgehen, dass die Strings nie `null` sind und dass der Integer grösser oder gleich 0 ist. In der Datei `EnthaltenMitAbstandTest.java` finden Sie ein paar einfache Tests. Wir empfehlen Ihre Lösung ausgiebig zu testen. **Tipp:** Lösen Sie die Aufgabe rekursiv.

Umkehrung (2018 W8)

Auf dem letzten Übungsblatt haben Sie eine Linked List für Integer implementiert. In dieser Aufgabe fügen Sie dieser `LinkedList` eine weitere Methode hinzu, welche die Liste umkehrt. Eine Liste gilt als umgekehrt, wenn für jedes Paar von Nodes `a` und `b`, für welche zuvor `a == b.next` gegolten hat, in der neuen (umgekehrten) Liste `b == a.next` gilt. Zusätzlich ist die erste Node der umgekehrten Liste, die letzte Node der ursprünglichen Liste und vice versa.

Vervollständigen Sie die Methode `reverse()` in der Klasse `LinkedList`. Die Methode soll, wie oben definiert, die Liste umkehren. Achten Sie darauf, dass Sie wirklich die Reihenfolge der Nodes selbst umkehren. Es reicht nicht aus, die Reihenfolge der enthaltenen `int`-Werte umzukehren. Es müssen auch in der umgekehrten Liste dieselben Instanzen von `IntNodes` wie in der ursprünglichen Liste verwendet werden. Erstellen Sie also *keine* neuen `IntNodes` per `new IntNode()`. In der Datei `UmkehrungTest.java` finden Sie einen einfachen Test.

Week 9

Square Grid (2023 W9)

In dieser Aufgabe betrachten wir gerichtete Graphen, wobei es für jeden Knoten g höchstens zwei gerichtete Kanten von g zu anderen Knoten f, h geben kann (f, g, h können gleich sein). Wir unterscheiden dabei zwischen der rechten und der unteren Kante (und damit dem rechten und dem unteren Knoten).

Die Klasse `Node` repräsentiert einen Knoten in einem solchen Graphen. Die Methode `Node.getRight()` (bzw. `Node.getDown()`) gibt den rechten Knoten (bzw. unteren Knoten) zurück (als `Node`-Objekt). Wenn der rechte Knoten von n_0 nicht existiert, dann gibt `Node.getRight()` `null` zurück (analog für den unteren Knoten). Die Methode `Node.setRight(Node r)` (bzw. `Node.setDown(Node d)`) setzt den rechten (bzw. unteren) Knoten.

Das Ziel der Aufgabe ist, einen von einem `Node`-Objekt definierten Graphen zu analysieren. Konkret geht es darum, die Grösse des grössten quadratischen Gitters in dem Graphen zu bestimmen, der mit dem übergebenen `Node`-Objekt beschrieben wird, welches den gleichen Ursprungsknoten wie der Graph hat.

Abbildung 5 zeigt ein Beispiel für einen Graphen mit Ursprungsknoten $n_{(0,0)}$ und Koordinaten $\{(0,0), (0,1), (1,0), (1,1)\}$, wobei jeweils `R` der rechten Kante und `D` der unteren Kante eines Knotens entspricht. Abbildung 6 zeigt zwei andere Graphen.

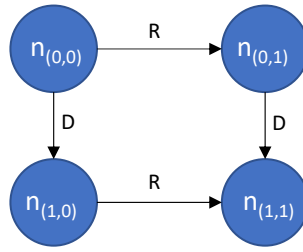


Abbildung 5: Graph als perfektes quadratisches Gitter

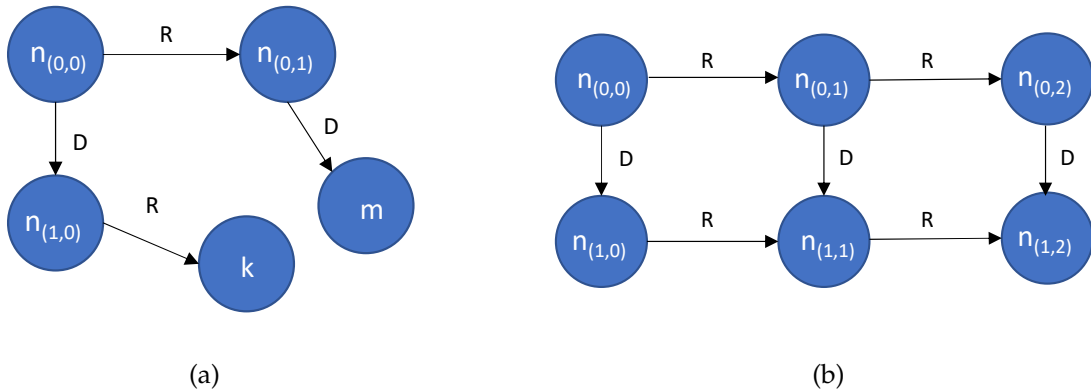


Abbildung 6: Graphen mit quadratischen Gittern als Teilgraphen

Ein Teilgraph G von einem Graphen G' (wie oben definiert) mit Ursprungsknoten u definiert ein quadratisches Gitter mit Ursprungsknoten u und Koordinaten K (wobei $K \subseteq \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0}$; das heisst, Koordinaten haben keine negativen Komponenten), so dass folgende Bedingungen gelten:

- Jeder Knoten in G ist auch in G' und jede Kante in G ist auch in G' .
- Jeder Knoten in G ist über die gerichteten Kanten vom Ursprungsknoten u erreichbar.
- Es gibt gleich viele Knoten in G wie Koordinaten in K . Ausserdem wird jede Koordinate $(i, j) \in K$ durch genau einen einzigartigen Knoten $n_{(i,j)}$ in G repräsentiert. Das heisst, wenn $\{(i, j), (i', j')\} \subseteq K$ und $(i, j) \neq (i', j')$, dann gilt $n_{(i,j)} \neq n_{(i',j')}$.
- Sei $(i, j) \in K$. Wenn der untere Knoten von $n_{(i,j)}$ existiert, dann gilt $(i+1, j) \in K$ und der untere Knoten von $n_{(i,j)}$ ist gegeben durch $n_{(i+1,j)}$. Wenn der rechte Knoten von $n_{(i,j)}$ existiert, dann gilt $(i, j+1) \in K$ und der rechte Knoten von $n_{(i,j)}$ ist gegeben durch $n_{(i,j+1)}$.
- Sei n die Grösse des Quadrats ($n \geq 1$). Dann ist K gegeben durch $K = \{(i, j) \mid 0 \leq i < n, 0 \leq j < n\}$.

Implementieren Sie die Methode `SquareGrid.analyzeSquareGrid(Node origin)`, welche die Grösse des *grössten* quadratischen Gitters in dem Graphen mit Ursprungsknoten `origin` (welches

origin als Ursprungsknoten hat) zurückgibt. Sie können davon ausgehen, dass origin nicht null ist (das bedeutet im Umkehrschluss, dass das kleinste Quadrat immer Grösse 1 hat).

In Abbildung 5 hat für den Ursprungsknoten $n_{(0,0)}$ das grösste Quadrat Grösse 2. In Abbildung 6a hat für den Ursprungsknoten $n_{(0,0)}$ das grösste Quadrat Grösse 1, und in Abbildung 6b hat für den Ursprungsknoten $n_{(0,0)}$ das grösste Quadrat ebenfalls Grösse 2 (das grösste Quadrat für den Ursprungsknoten $n_{(0,1)}$ hat hier ebenfalls Grösse 2; für alle anderen Ursprungsknoten in dem Graphen hat das grösste Quadrat Grösse 1).

In der Klasse Main finden Sie eine main-Methode, welche Sie verwenden können, um Ihre Implementierung zu testen. In der Datei "SquareGridTest.java" finden Sie ausserdem einige Tests.

Klassen Rätsel (2022 W9)

In dieser Aufgabe sollen Sie zeigen, dass Sie mit Klassen und Vererbung umgehen können. Im Anhang 34.1 finden Sie ein Programm, welches Instanzen von Klassen erstellt und Methoden aufruft. Das Programm macht nichts sinnvolles und dient nur dem Testen ihrer Fähigkeiten. In Anhang 34.2 befinden sich die verwendeten Klassen, jedoch sind die Klassen noch nicht vollständig. Bei manchen der Klassen fehlt noch die "extends" Klausel, welche angibt, dass eine Klasse von einer anderen Klasse erbt. Ihre Aufgabe ist es die nötigen "extends" Klauseln hinzuzufügen, sodass alles kompiliert und sodass die Ausgabe des Programms von Anhang 34.1 am Ende so aussieht, wie im Anhang 34.3 gezeigt.

Der Code von Anhang 34.1 and Anhang 34.2 befindet sich in ihrem Source Ordner. Zusätzlich enthält "KlassenTest.java" einen Unit-Test, welcher prüft, ob die Ausgabe des Programms dem Output aus Anhang 34.3 entspricht. Beachten Sie, dass Sie für diese Aufgabe **ausschliesslich** "extends" Klauseln hinzufügen (diese kann es nur an den grauen Boxen aus Anhang 34.2 geben), kein anderer Code darf verändert werden. Wenn wir bei der Korrektur feststellen, dass anderer Code geändert wurde, dann können Sie 0 Punkte erhalten.

In dieser Aufgabe können Sie auch Teilpunkte erhalten, falls manche der Klassen aus Anhang 34.2 nicht kompilieren.

Tipp: Lösen Sie die Aufgabe zuerst auf Papier, ohne die Hilfe von Eclipse. Sobald Sie herausgefunden haben, welche Klassen von welchen Klassen erben, testen Sie Ihre Lösung in Eclipse. Dies hilft Ihnen, Ihr Wissen über Vererbung zu testen. In der Vergangenheit wurden ähnliche Aufgaben im schriftlichen Teil der Prüfung gestellt.

Ballspiel (2021 W9)

In dieser Aufgabe implementieren Sie ein Spiel mit Bällen für einen Spieler. In einer Urne befinden sich farbige Bälle. Zusätzlich hat man einen Punktestand. Ein Spiel besteht aus N Runden. Pro Runde wird: (1) ein Ball aus der Urne genommen, dann (2) wird abhängig von der Farbe des Balls eine Aktion ausgeführt, dann (3) wird ein Sticker auf den Ball geklebt (damit man weiss wie oft ein Ball schon gezogen wurde), und zuletzt (4) wird der Ball wieder in die Urne getan. Die folgende Auflistung beschreibt welche Aktionen bei welcher Farbe ausgeführt werden:

- **Grün:** Addiere zu dem Punktestand die Anzahl der Sticker auf dem gezogenen Ball.

- **Blau:** Subtrahiere von dem Punktestand die Anzahl der Sticker auf dem gezogenen Ball. Lege zusätzlich einen neuen (ohne Sticker) blauen Ball in die Urne.
- **Rot:** Multipliziere den Punktestand mit 2. Falls es noch mindestens einen weiteren Ball in der Urne hat, dann ziehe einen zusätzlichen Ball *b* aus der Urne und entferne diesen Ball *b* aus dem Spiel. Für *b* wird keine Aktion ausgeführt.
- **Pink:** Führe die gleiche Aktion aus wie für Rot. Lege zusätzlich einen neuen (ohne Sticker) habsburgergelben Ball in die Urne.
- **Habsburgergelb:** Multipliziere den Punktestand mit der Anzahl der Sticker auf dem gezogenen Ball und addiere nach der Multiplikation noch 1 zum Punktestand. Zusätzlich wird der Effekt von dem Ball, welcher 3 Züge später gezogen wird, nicht ausgeführt.

Implementieren Sie die Methode `BallGame.playGame(int turns, int startScore)`. Die Methode gibt den Punktestand zurück, welcher erreicht wird, wenn das Spiel mit einem Startpunktestand von `startScore` über `turns` viele Runden gespielt wird. Die Urne wird als Argument zu dem Konstruktor von `BallGame` gegeben. Die Klasse `BallPool` implementiert die Urne. Die Klasse `Ball` implementiert einen generischen Ball und wird von `GreenBall`, `BlueBall`, `RedBall`, und `PinkBall` für die jeweils gefärbten Bälle erweitert. Den habsburgergelben Ball sollen Sie selber implementieren. Ein Ball muss mit der Methode `BallPool.draw()` aus der Urne gezogen werden und muss mit der Methode `BallPool.add(Ball ball)` in die Urne getan werden. Die Methode `BallPool.draw()` gibt `null` zurück, falls die Urne leer ist. Wir verwenden die Methode `BallPool.add(Ball ball)` auch um die Urne am Anfang mit Bällen zu befüllen. Sie dürfen annehmen, dass die Urne am Anfang vom Spiel nie leer ist. Die Datei `"BallGameTest.java"` enthält Tests. Sehen Sie sich diese Datei an um Beispiele zu sehen. **Achtung:** Für die Korrektur werden wir eine andere Implementation von `BallPool` verwenden. Sie sollten `BallPool` weder ändern noch erweitern. Sie sollten nur die `draw` und `add` Methode der Klasse `BallPool` verwenden. Sie sollten keine anderen Methoden oder Attribute von `BallPool` verwenden, da diese in der anderen Implementation nicht das Gleiche sein werden. Die Tests in `"BallGameTest.java"` verwenden ebenfalls eine andere Implementation von `BallPool`. Damit stellen wir sicher, dass nicht die Urne selber modifiziert wird.

Klassen Rätsel (2020 W9)

In dieser Aufgabe sollen Sie zeigen, dass Sie mit Klassen und Vererbung umgehen können. Im Anhang 34.1 finden Sie ein Programm, welches Instanzen von Klassen erstellt und Methoden aufruft. Das Programm macht nichts sinnvolles und dient nur dem Testen ihrer Fähigkeiten. In Anhang 34.2 befinden sich die verwendeten Klassen, jedoch sind die Klassen noch nicht vollständig. Bei manchen der Klassen fehlt noch die `"extends"` Klausel, welche angibt, dass eine Klasse von einer anderen Klasse erbt. Ihre Aufgabe ist es die nötigen `"extends"` Klauseln hinzuzufügen, sodass alles kompiliert und sodass die Ausgabe des Programms von Anhang 34.1 am Ende so aussieht, wie im Anhang 34.3 gezeigt.

Der Code von Anhang 34.1 and Anhang 34.2 befindet sich in ihrem Source Ordner. Zusätzlich enthält `"KlassenTest.java"` einen Unit-Test, welcher prüft, ob die Ausgabe des Programms dem Output aus Anhang 34.3 entspricht. Beachten Sie, dass Sie für diese Aufgabe **ausschliesslich**

“extends” Klauseln hinzufügen (diese kann es nur an den grauen Boxen aus Anhang 34.2 geben), kein anderer Code darf verändert werden. Wenn wir bei der Korrektur feststellen, dass anderer Code geändert wurde, dann können Sie 0 Punkte erhalten.

In dieser Aufgabe können Sie auch Teilpunkte erhalten, falls manche der Klassen aus Anhang 34.2 nicht kompilieren.

Tipp: Lösen Sie die Aufgabe zuerst auf Papier, ohne die Hilfe von Eclipse. Sobald Sie herausgefunden haben, welche Klassen von welchen Klassen erben, testen Sie Ihre Lösung in Eclipse. Dies hilft Ihnen Ihr Wissen über Vererbung zu testen. In der Vergangenheit wurden ähnliche Aufgaben im schriftlichen Teil der Prüfung gestellt.

Anhang: Testprogramm Bonusaufgabe

```
public class Klassen {  
  
    ...  
  
    public static void klassen(PrintStream output) {  
  
        KlasseZ a = new KlasseA();  
        KlasseB b = new KlasseB();  
        KlasseB c = new KlasseC();  
        KlasseB d = new KlasseD();  
        KlasseZ e = new KlasseE();  
        KlasseB f = new KlasseF();  
  
        if (((KlasseA)a).foo() && ((KlasseA)e).foo()) {  
            output.println("Zuerich");  
        } else {  
            output.println("Bern");  
        }  
  
        output.println(c.foo() + "␣" + b.foo());  
        output.println(f.foo() + "␣" + c.foo());  
        if ((f instanceof KlasseD) || (d instanceof KlasseF)) {  
            output.println("Related") ;  
        } else {  
            output.println("Unrelated");  
        }  
        output.println(d.foo() + "␣" + c.foo());  
    }  
}
```

Anhang: Klassen Bonusaufgabe

```
class KlasseZ  { }  
  
class KlasseA  {  
    boolean foo() {
```

```

        return true;
    }
}

class KlasseB [REDACTED] {
    int x = 0;

    int foo() {
        return x;
    }
}

class KlasseC [REDACTED] {
    int foo() {
        return x+1;
    }
}

class KlasseD [REDACTED] {
    int foo() {
        int y = super.foo();
        x = y+1;
        return x;
    }
}

class KlasseE [REDACTED] {
    boolean foo() {
        return false;
    }
}

class KlasseF [REDACTED] {
    int foo() {
        int y = super.foo();
        x = y+1;
        return x;
    }
}

```

Anhang: Ausgabe Bonusaufgabe

```

Bern
1 0
1 1
Unrelated
2 1

```

Klassen Rätsel (2019 W9)

In dieser Aufgabe sollen Sie zeigen, dass Sie mit Klassen und Vererbung umgehen können. Im Anhang 35.1 finden Sie ein Programm, welches Instanzen von Klassen erstellt und Methoden aufruft. Das Programm macht nichts sinnvolles und dient nur dem Testen ihrer Fähigkeiten. Die Ausgabe des Programms soll am Ende aussehen, wie im Anhang 35.2 gezeigt. In der Datei "Klassen.java" finden Sie das Programm aus Anhang 35.1. Zusätzlich enthält "KlassenTest.java" einen Unit-Test, welcher prüft, ob der Output des Programms dem Output aus Anhang 35.2 entspricht.

Erstellen und implementieren Sie alle Klassen und Methoden, welche dafür notwendig sind, dass das Programm kompiliert und den korrekten Output ausgibt. Diese Klassen müssen sich in einer oder mehreren zusätzlichen Datei(en) (im "src"-Ordner) befinden, *nicht* in der "Klassen.java"-Datei. Die "Klassen.java"-Datei dürfen Sie *nicht* verändern. In dieser Aufgabe können Sie auch Teilpunkte für eine nicht compilierende "Klassen.java"-Datei erhalten. Beachten Sie jedoch, dass die Datei(en), in welchen sich Ihre Klassen befinden, keinerlei Kompilierfehler enthalten dürfen, da Sie sonst keine Teilpunkte erhalten.

Tipp: Konzentrieren Sie sich zuerst darauf, dass das Programm kompiliert *ohne* den Output zu beachten. Passen Sie danach die Implementierung an, damit der Output passt.

Anhang: Testprogramm Bonusaufgabe

```
public class Klassen {  
  
    ...  
  
    public static void klassen(PrintStream output) {  
  
        Lambda l = new Lambda();  
        Sigma s = new Sigma();  
        Alpha a = new Alpha();  
        Kappa k = new Kappa();  
        Iota i = new Iota();  
        Zeta z = new Zeta();  
        Beta b = new Beta();  
        Omega o = new Omega();  
  
        Omega[] os = {z, k, l, (Omega) i};  
        for (int j = 0; j < os.length; j += 1) {  
            output.println(os[j].name());  
            output.println("---");  
        }  
  
        b = doSomething(l);  
        int n = choose(l, z);  
  
        b = l.choose(a, s, i);  
        output.println(b == a);  
        output.println("---");  
    }  
}
```

```

        boolean c = s.calc() > i.calc().length() && z.calc();

        Lambda ll = o.create();
        output.println(ll.name());
        output.println("---");
    }

    public static int choose(Kappa k, Alpha a) {
        return 0;
    }

    public static Kappa choose(Alpha a, Kappa k) {
        return a;
    }

    public static Sigma doSomething(Iota i) {
        return null;
    }

    public static int doSomething(Kappa k) {
        return 0;
    }
}

```

Anhang: Ausgabe Bonusaufgabe

```

Zeta
---
Kappa
---
Lambda
---
Iota
---
true
---
Kappa
---

```

Klassen Rätsel (2018 W9)

In dieser Aufgabe sollen Sie zeigen, dass Sie mit Klassen und Vererbung umgehen können. Im Anhang 36.1 finden Sie ein Programm, welches Instanzen von Klassen erstellt und Methoden aufruft. Die Ausgabe des Programms soll am Ende aussehen, wie im Anhang 36.2 gezeigt. In der Datei "Klassen.java" finden Sie das Programm aus Anhang 36.1. Zusätzlich enthält "KlassenTest.java"

einen Unit-Test, welcher prüft, ob der Output des Programms dem Output aus Anhang 36.2 entspricht.

Erstellen und implementieren Sie alle Klassen und Methoden, welche dafür notwendig sind, dass das Programm kompiliert und den korrekten Output ausgibt. Diese Klassen müssen sich in einer oder mehreren zusätzlichen Dateien (im "src"-Ordner) befinden, *nicht* in der "Klassen.java"-Datei. Die "Klassen.java"-Datei dürfen Sie *nicht* verändern. Beachten Sie, dass die Dateien, in welchen sich Ihre Klassen befinden, keinerlei Kompilierfehler enthalten dürfen; andernfalls können Sie keine Teilpunkte erhalten.

Tipp: Konzentrieren Sie sich zuerst darauf, dass das Programm kompiliert *ohne* den Output zu beachten. Passen Sie danach die Implementierung an, damit der Output passt.

Testprogramm Bonusaufgabe

```
public class Klassen {
    ...

    public static void klassen(PrintStream output) {
        Lambda l = new Lambda();
        Sigma s = new Sigma();
        Alpha a = new Alpha();
        Kappa k = new Kappa();
        Iota i = new Iota();
        Zeta z = new Zeta();

        Delta[] ds = {s, a, k};
        for (int j = 0; j < ds.length; j += 1) {
            output.println(ds[j].name());
            output.println("---");
        }
        output.println(i.name());
        output.println("---");

        choose(z, i, z);
        Alpha aa = l.choose(s, a, k);

        Iota ii = k.create();
        output.println(ii.name());

        if (k == aa) {
            System.out.println("ok");
        } else if (i.calc() > k.calc().length() && s.calc()) {
            System.out.println("wrong");
        } else {
            System.out.println("wrong");
        }

        Zeta zz = z.create();
        output.println(zz.name());
    }
}
```

```

        public static Delta choose(Lambda l, Iota i, Alpha a) {
            return i;
        }
    }
}

```

Ausgabe Bonusaufgabe

```

Sigma
---
Alpha
---
Kappa
---
Iota
---
Gamma
ok
ahplA

```

Week 10

Datenbanken (2023 W10)

In dieser Aufgabe implementieren Sie für eine Datenbank von Personengesundheitsdaten das Deklassifizieren von Einträgen (Task a) und das Verlinken von Einträgen (Task b). Alle Unteraufgaben können separat gelöst werden.

Die Datenbank selber ist bereits mit der Klasse `Database` implementiert. Die Datenbank hält eine Liste von Einträgen, welche durch die Klasse `Item` repräsentiert werden. Die folgenden 4 Paragraphen erklären alle in der Vorlage gegebenen Klassen im Detail.

Item Die Klasse `Item` repräsentiert einen Datenbankeintrag mit 4 Attributen: eine ID (int), ein Alter (int), einen Gesundheitswert (int), und ein Sicherheitslevel, welches durch die Klasse `Level` repräsentiert wird. Alter und Gesundheitswert sind immer ≥ 0 . Die Methoden `Item.getID()`, `Item.getAge()`, `Item.getHealth()`, `Item.getLevel()` geben jeweils die ID, das Alter, den Gesundheitswert, und das Sicherheitslevel eines Eintrags zurück. Die Methode `Item.setHealth(int newHealth)` setzt den Gesundheitswert auf `newHealth`. Die anderen Attribute können nicht geändert werden.

Level Die Klasse `Level` repräsentiert ein Sicherheitslevel. Ein Sicherheitslevel wird über eine Liste von Integern definiert, welches in einem Attribut der Klasse `Level` gespeichert wird und von der Methode `Level.getPoints()` zurückgegeben wird. Ein Level A ist *verwandt* mit einem Level B, falls die Summe der Werte in `A.getPoints()` gleich der Summe der Werte in `B.getPoints()` ist. Zum Beispiel ist das Level `[1,2,3,4]` verwandt mit den Levels `[10]` und `[4,6]` (die Summe ist überall 10), aber nicht mit dem Level `[4,5]`.

ItemFactory Die Klasse `ItemFactory` wird verwendet, um Datenbankeinträge zu erstellen. Die Methode `ItemFactory.createItem(Level level, int id, int age, int health)` gibt ein Exemplar der Klasse `Item` zurück, deren Attribute mit den Argumenten initialisiert wurden.

Database Die Klasse `Database` repräsentiert eine Datenbank und hat folgende vorgegebene Methoden:

- `Database.getItemFactory()` gibt ein Exemplar von `ItemFactory` zurück. Die `ItemFactory` `I` ist assoziiert mit der Datenbank `D`, falls `I` von `D.getItemFactory()` zurückgegeben wird.
 - `Database.add(Item item)` fügt der Datenbank den Eintrag `item` hinzu.
 - `Database.getItems()` gibt die Liste aller Einträge zurück, welcher der Datenbank hinzugefügt wurden. Sie dürfen annehmen, dass für eine Datenbank `D` alle Einträge in `D.getItems()` eine einzigartige ID haben, über `D.add` hinzugefügt wurden, über `D.getItemFactory()` erstellt wurden, und keiner anderen Datenbank hinzugefügt werden. Ein hinzugefügter Eintrag wird nie wieder entfernt.
- a) Implementieren Sie die Methode `ItemFactory.createDeclass(Level level, int id, int targetId)`, die einen *Deklassifikationseintrag* zurückgibt. Ein Deklassifikationseintrag ist selber ein Eintrag, also ein Exemplar der Klasse `Item`. Ein Deklassifikationseintrag hat damit auch eine ID, ein Sicherheitslevel, ein Alter, und einen Gesundheitswert, welche von den entsprechenden getter-Methoden zurückgegeben werden. ID und Sicherheitslevel eines Deklassifikationseintrags sind jeweils das `id` und `level` Argument des `createDeclass` Aufrufs, mit welchem der Eintrag erstellt wurde. Das Alter und der Gesundheitswert eines Deklassifikationseintrags sind jeweils das Alter und der Gesundheitswert des *Zieleintrags* vom Deklassifikationseintrag. Der Zieleintrag von einem Deklassifikationseintrag `D` ist der Eintrag `E`, so dass
- `E.getID()` gleich dem Parameter `targetId` ist, mit welchem `D` erstellt wurde; *und*
 - `E` aus der Datenbank ist, mit welcher die `ItemFactory` assoziiert ist, mit welcher `D` erstellt wurde.

Falls es keinen Zieleintrag gibt, wird eine `IllegalArgumentException` von der Methode `createDeclass` geworfen. Beachten Sie, dass Zieleinträge selber Deklassifikationseinträge sein können. Ein Aufruf der Methode `Item.setHealth(h)` auf einem Deklassifikationseintrag hat keinen Effekt; dies wird nicht in den Tests überprüft.

Ein Deklassifikationseintrag `R` *erreicht* einen Eintrag `A`, falls entweder `A` der Zieleintrag von `R` ist oder falls der Zieleintrag von `R` ein Deklassifikationseintrag ist, welcher `A` erreicht. Die Methode `createDeclass` wirft eine `IllegalArgumentException`, falls der zurückzugebene Deklassifikationseintrag `R` einen Eintrag erreicht, dessen Level verwandt ist mit dem Level von `R`. Zur Erinnerung: Der Paragraph über die Klasse `Level` erklärt, wann zwei Level verwandt sind.

- b) Implementieren Sie die Methode `Database.createLink(List<Integer> ids)`. Der Methodenaufruf `D.createLink(ids)` *verlinkt* alle Einträge der Datenbank `D` miteinander, welche eine ID haben, die im Argument `ids` enthalten ist. Wenn `E.setHealth(h)` auf einem Eintrag `E`

aufgerufen wird, dann wird der Gesundheitswert aller Einträge, welche mit E verlinkt sind, auf das Argument `h` gesetzt. Einträge können beliebig oft verlinkt werden und verlinken ist transitiv, das heisst, wenn ein Eintrag A mit einem Eintrag B verlinkt ist und B mit einem Eintrag C verlinkt ist, dann ist A auch mit C verlinkt. Verlinken ist auch immer symmetrisch, das heisst, wenn A mit B verlinkt ist, dann ist auch B mit A verlinkt. Zusätzlich ist verlinken reflexiv, das heisst, ein Eintrag ist immer mit sich selber verlinkt.

Der Aufruf `D.createLink(ids)` soll eine `IllegalArgumentException` werfen, falls es eine ID im Argument `ids` gibt, für welche es keinen Eintrag mit der gleichen ID in der Datenbank D gibt.

Wir geben zwei Testdateien zur Verfügung. “DatabaseTest.java” enthält Tests, welche wir an einer Prüfung geben würden. “GradingDatabaseTest.java” enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie Ihre Lösung zuerst ausgiebig mit “DatabaseTest.java” (am besten fügen Sie selber neue Tests hinzu) und dann können Sie “GradingDatabaseTest.java” verwenden, um zu sehen wie Ihre Lösung an einer Prüfung abgeschnitten hätte.

Equivalent Executable Graphs (2022 W10)

In dieser Aufgabe verwenden wir gerichtete azyklische Graphen, um Programme zu repräsentieren. Knoten in Graphen werden durch `Node`-Objekte modelliert. `Node.getSubnodes()` gibt die Kinderknoten als ein Array zurück (m ist genau dann ein Kinderknoten von n , wenn es eine ausgehende gerichtete Kante von n zu m gibt). Wir unterscheiden zwei Arten von Knoten, wobei die Methode `Node.getType()` die Knotenart als String zurückgibt. Wir beschreiben nun die zwei Knotenarten:

1. Additionsknoten (`Node.getType()` ist “ADD”): Solche Knoten besitzen einen Additionswert a gegeben durch `Node.getValue()` (eine ganze Zahl). Sie dürfen annehmen, dass ein Additionsknoten keine Kinderknoten hat.
2. Sequenzknoten (`Node.getType()` ist “SEQ”): Sie dürfen davon ausgehen, dass Sequenzknoten immer mindestens einen Kinderknoten haben. `Node.getValue()` ist irrelevant.

Sie dürfen annehmen, dass es zwischen zwei Knoten immer höchstens einen Pfad gibt.

Implementieren Sie die Methode `GraphExecution.isSubProgram(Node n1, Node n2)`, welche entscheidet, ob $n2$ ein *Unterprogramm* von $n1$ ist. Der Knoten $n2$ ist genau dann ein Unterprogramm von $n1$, wenn es einen Knoten $n1'$ gibt, welcher von $n1$ erreichbar ist (das heisst, $n1'$ ist gleich $n1$ oder kann durch die gerichteten Kanten im Graphen von $n1$ aus erreicht werden), so dass $n1'$ und $n2$ äquivalent sind.

Wir beschreiben nun, wann zwei Knoten äquivalent sind. Zwei Additionsknoten sind genau dann äquivalent, wenn ihre Additionswerte gleich sind. Zwei Sequenzknoten n und m sind genau dann äquivalent, wenn sie die gleiche Anzahl Kinderknoten haben und wenn es ein Objekt `nSubPerm` vom Typ `Node[]` gibt, welches eine Permutation von `n.getSubnodes()` ist, so dass `nSubPerm[i]` und `m.getSubnodes()[i]` äquivalent sind für $0 \leq i < \text{nSubPerm.length}$.

Bemerkung zur Permutation: Seien `ns1` und `ns2` Objekte vom Typ `Node[]`. `ns1` ist eine Permutation von `ns2` genau dann, wenn `ns1.length == ns2.length` und es gibt eine injektive (d.h.

jeder Input wird einem anderen Wert zugewiesen) Funktion $f : \{0, 1, \dots, ns1.length - 1\} \rightarrow \{0, 1, \dots, ns1.length - 1\}$, so dass: $ns1[i] == ns2[f(i)]$ (für $i \in \{0, 1, \dots, ns1.length - 1\}$).

Wir geben zwei Testdateien zur Verfügung. "GraphExecutionTest.java" enthält Tests, welche wir an einer Prüfung geben würden. "GradingGraphExecutionTest.java" enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie Ihre Lösung zuerst ausgiebig mit "GraphExecutionTest.java" (am besten fügen Sie selber neue Tests hinzu) und dann können Sie "GradingGraphExecutionTest.java" verwenden, um zu sehen wie Ihre Lösung an einer Prüfung abgeschnitten hätte.

Flex Array (2021 W10)

Sie sollen einen Datentyp implementieren, der flexible Arrays von ints zulässt. Flexibel heisst in diesem Zusammenhang, dass die Basis des Arrays frei gewählt werden kann (sofern es sich um einen int handelt). Ein Array mit Basis *base* und Länge *length* sieht dann so aus:

Index	<i>base</i>	<i>base+1</i>	<i>base+2</i>				<i>base + length-1</i>
Wert	0	0	0				0

Legale Indices für diesen Array sind *base*, *base+1*, ..., *base+length-1*.

Der Array kann int Werte speichern zwischen `Integer.MIN_VALUE+1` und `Integer.MAX_VALUE`. Sie können davon ausgehen, dass alle Tests nur Werte innerhalb dieses Intervalls in den Arrays speichern. Der Wert `Integer.MIN_VALUE` wird verwendet, um Fehler anzuzeigen.

Die Aufgabe besteht aus zwei Teilen. Sie sollten beide Teile lösen, wenn Sie die volle Punktzahl erreichen wollen. Der erste Teil ist leichter als der zweite Teil, aber dennoch eine gute Übung. (Wir haben die Aufgabe in zwei Teile zerlegt damit Sie üben können, mit gegliederten Aufgaben umzugehen.)

Sie dürfen für diese Aufgabe nichts aus `java.util` oder anderen Paketen verwenden. Das heisst auch, dass alle Imports verboten sind. Falls Ihr Code etwas importiert, auch wenn es nicht verwendet wird, dann bekommen Sie 0 Punkte.

1. Implementieren Sie die Klasse `FlexArray`. Vervollständigen Sie die Klasse im Template und implementieren Sie die folgenden Methoden und Konstruktoren:

- `FlexArray(int length, int base)` erstellt einen `FlexArray` wie oben gezeigt. Die Elemente des Arrays werden mit 0 initialisiert. Sie können davon ausgehen, dass *base+length* als int dargestellt werden kann und dass *length* ≥ 0 ist.
- `FlexArray(int length)` erstellt einen `FlexArray` für *length* viele Elemente mit Basis 0. Die Elemente des Arrays werden mit 0 initialisiert.
- `FlexArray()` erstellt einen `FlexArray` ohne Platz für Elemente.
- `FlexArray.read(int index)` gibt den Wert zurück, der für den Index *index* gespeichert wird, falls es sich um einen legalen Index handelt. Falls das Argument *index* nicht legal ist (also das entsprechende Element im Array nicht existiert), dann gibt die Methode `Integer.MIN_VALUE` zurück.

- `FlexArray.write(int index, int value)` speichert den Wert `value` im Element mit dem Index `index`, falls `index` ein legaler Index ist. Andernfalls hat diese Methode keine Wirkung.
 - `FlexArray.rawArray()` gibt einen neuen `int[]` Array zurück, der die im `FlexArray` gespeicherten Werte (in der richtigen Reihenfolge) enthält.
 - `FlexArray.base()` und `FlexArray.length()` liefern jeweils die Basis und Länge eines `FlexArrays`.
 - `FlexArray.equals(Object o)` überschreibt die Methode `equals` von `Object` und gibt `true` zurück, falls der Parameter auf einen `FlexArray` verweist, der die selbe Basis, selbe Länge hat, und selben Elemente wie der Empfänger hat. Andernfalls gibt die Methode `false` zurück.
 - `FlexArray.copy()` liefert eine Kopie eines `FlexArrays`. (Kopien können unabhängig von einander verändert werden.)
2. Implementieren Sie die Methode `FlexArray.merge(FlexArray f)`. Für zwei `FlexArrays` `g` und `f` ist die Operation `g.merge(f)` nur definiert, wenn die Index Bereiche überlappen und `f` und `g` eine Länge ≥ 1 haben. Ist das nicht der Fall, dann gibt die Methode `null` zurück, sonst einen neuen `FlexArray`. Der neue `FlexArray` ist wie folgt bestimmt: Wenn der Index `I` nur für `g` ein legaler Index ist, so ist im Ergebnisarray der Wert für den Index `I` der Wert aus `g`. Wenn der Index `I` nur für `f` ein legaler Index ist, so ist im Ergebnisarray der Wert für den Index `I` der Wert aus `f`. Wenn der Index `I` für `g` und `f` ein legaler Index ist, so ist im Ergebnisarray der Wert für den Index `I` die Summe der Werte aus `g` und `f`.

Die Datei "FlexArrayTest.java" enthält verschiedene Beispiele.

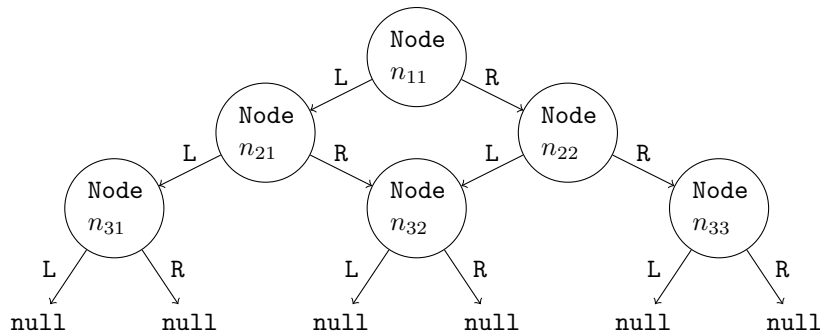
Pyramide (2020 W10)

Die Klasse `Node` repräsentiert einen Knoten in einem gerichteten Graphen, wobei es für jeden Knoten n_1 höchstens zwei gerichtete Kanten von n_1 zu anderen Knoten n_2, n_3 geben kann (n_2 und n_3 können gleich sein). Wir unterscheiden dabei zwischen dem linken und dem rechten Knoten. Die Methode `Node.getLeft()` gibt den linken Knoten und `Node.getRight()` den rechten Knoten zurück (als `Node`-Objekt). Wenn der linke Knoten von n_1 nicht existiert, dann gibt `Node.getLeft()` `null` zurück (analog für den rechten Knoten).

Das Ziel dieser Aufgabe ist, für ein `Node`-Objekt zu entscheiden, ob der durch das `Node`-Objekt definierte Graph einer Pyramide entspricht. Zum Beispiel entspricht der folgende Graph einer Pyramide.

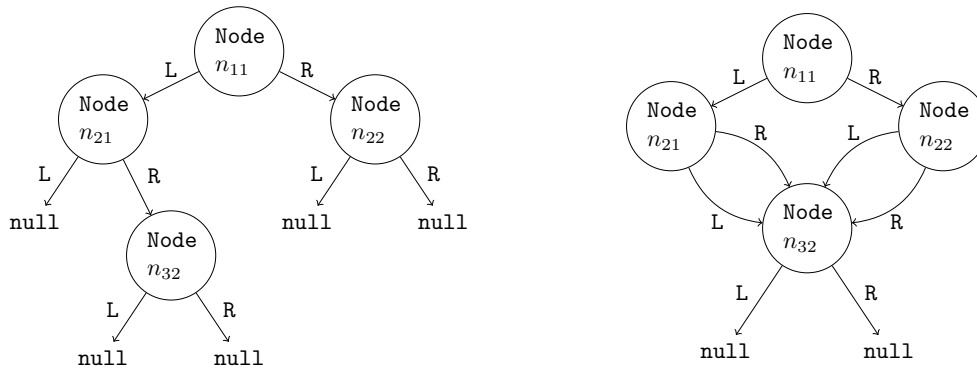
Beachten Sie, dass der rechte Knoten von n_{21} gleich ist wie der linke Knoten von n_{22} (das heisst die `Node`-Objekte sind gleich!). Ein Graph (wie oben repräsentiert) definiert eine Pyramide genau dann, wenn folgende Bedingungen gelten:

- Der Graph kann in $k \geq 1$ Stufen (Stufe 1, Stufe 2,..., Stufe k) aufgeteilt werden, wobei Stufe i aus i unterschiedlichen Knoten $n_{i1}, n_{i2}, \dots, n_{ii}$ besteht. Falls der Graph k Stufen hat, dann hat dieser genau $\frac{k(k+1)}{2}$ unterschiedliche Knoten (Knoten aus verschiedenen Stufen sind unterschiedlich).



- Für Stufe i ($1 \leq i < k$) gilt: der linke Knoten von n_{ij} ($1 \leq j \leq i$) ist durch $n_{(i+1)j}$ gegeben und der rechte Knoten von n_{ij} ist durch $n_{(i+1)(j+1)}$ gegeben.
- Für Stufe k gilt: es gibt keinen linken und keinen rechten Knoten für n_{kj} ($1 \leq j \leq k$).

Die folgenden Graphen entsprechen zum Beispiel keinen Pyramiden:



Implementieren Sie die boolean `isPyramid(Node node)`-Methode, welche, für den Graph G durch `node` definiert, entscheidet, ob G eine Pyramide definiert. Sie dürfen annehmen, dass G keine Zyklen hat. Die Methode soll eine `IllegalArgumentException` werfen, wenn das Argument `null` ist.

Tipp: Prüfen Sie die Bedingungen Stufe für Stufe, beginnend bei Stufe 1.

Wahlen (2019 W10)

In der letzten Bonusaufgabe sollten Sie zeigen, dass Sie mit der Theorie von Klassen und Vererbung umgehen können. In dieser Aufgabe geht es um die praktische Anwendung von Klassen und Interfaces. Sie werden ein Programm vervollständigen, welches Wahlergebnisse eines Kantons auswertet nach dem [Doppeltproportionalen Zuteilungsverfahren](#) (Wir geben weiter unten eine kurze Beschreibung). Ein Grossteil des Zuteilungsverfahrens ist bereits implementiert und wird von der Methode `wahlauswertung` aus der Datei "Wahlen.java" aufgerufen. Die Methode nimmt als Parameter einen zweidimensionalen Array `stimmen` (`stimmen[i][j]` enthält die Anzahl gezählter Stimmen aus dem Wahlkreis mit dem Index i für die Partei mit dem Index j), einen

eindimensionalen Array `sitzeProWahlkreis` (`sitzeProWahlkreis[i]` enthält die Anzahl verfügbarer Sitze für den Wahlkreis `i`), eine Zuteilungsstrategie `verteilung` und eine Rundungsstrategie `runder`. Die Methode gibt einen zweidimensionalen Array zurück, welcher an der Stelle `[i][j]` für den Wahlkreis mit Index `i` die Sitze der Partei mit Index `j` enthält. Jeder Wahlkreis hat die gleiche Anzahl Parteien, wobei sowohl die Anzahl Wahlkreise als auch die Anzahl Parteien durch die beiden Dimensionen des Arrays stimmen gegeben sind.

Ihre Aufgabe ist es, den restlichen Code an den fehlenden Stellen hinzuzufügen, welche mit `TODO` markiert sind. An jeder dieser Stellen ist explizit geschrieben was verlangt wird. Man kann die Aufgabe mit ungefähr 30 Zeilen Code lösen. Die Hauptherausforderung ist es, mit Interfaces umzugehen und nicht von grösserem Code überwältigt zu werden. Sie können neue Interfaces, Klassen, Methoden und Felder hinzufügen, jedoch dürfen Sie keine der gegebenen Deklarationen ändern (ausser wenn es explizit geschrieben ist). Sie dürfen einer Klasse oder einem Interface immer weitere Interfaces oder Klassen in der `extends` oder `implements` Klausel hinzufügen.

Die Datei `WahlenTest.java` enthält Tests, um Ihre Lösung zu testen. Das Verfahren liefert nicht für alle Inputs eine Lösung. In der Realität würde eine Kommission Uneindeutigkeiten auflösen. Für uns reicht es jedoch in diesen Fällen `null` zurückgegeben. Sie müssen das Verfahren nicht anpassen und können davon ausgehen, dass nur Inputs mit einem eindeutigen Ergebnis zum testen verwendet werden.

Nun zur Erklärung des Verfahrens: Wir geben hier eine kürzere Erklärung; bei Fragen folgen Sie dem oberen Link (enthält auch Beispiele) oder sehen Sie sich den zur Verfügung gestellten Code und die Tests an. Das Doppeltproportionale Zuteilungsverfahren ist in zwei Phasen unterteilt, die Oberzuteilung und die Unterzuteilung. In der Oberzuteilung werden jeder Partei eine Anzahl Sitze zugeteilt. Dafür werden zuerst für jede Partei die Summe aller Stimmen über alle Wahlkreise berechnet. Zusätzlich wird die Summe k aller gesamten verfügbaren Sitze berechnet. Danach wird ein sogenanntes [Höchstzahlverfahren](#) angewendet: Jede Zahl der Stimmen einer Partei wird durch eine Reihe von Divisoren geteilt (zum Beispiel 0.5, 1.5, 2.5, 3.5, ...), wobei die Ergebnisse aufgelistet werden. Eine Partei erhält dann i Sitze, wenn die Partei i der k grössten aufgelisteten Einträge hat. Wir parametrisieren die Oberzuteilung mit einer `Sitzzuteilungsstrategie`, welche die verwendete Reihe der Divisoren vorgibt und in der `wahlauswertung` Methode als Parameter übergeben wird. Definieren Sie hierfür Methoden in dem Interface `Sitzzuteilungsstrategie` und implementieren Sie zwei Klassen welche das [Sainte-Laguë-Verfahren](#) (Divisoren Reihe 0.5, 1.5, 2.5, 3.5, ...) und das [D'Hondt-Verfahren](#) (Divisoren Reihe 1, 2, 3, 4, ...) implementieren. Zusätzlich implementieren Sie die Methoden `getSainteLague` und `getDHondt` der `SitzzuteilungsstrategieFactory` Klasse, damit wir in unseren Tests die entsprechenden Strategien instanzieren können. All diese Anweisungen stehen auch im Code, mit `TODO` markiert.

Bei der Unterzuteilung werden nach der Oberzuteilung die genauen Sitze jeder Partei für jeden Wahlkreis zugeordnet. Es ist ein iteratives Verfahren, welches angewendet wird, bis eine Lösung gefunden wird. Zuerst wird für jeden Wahlkreis ein Divisor gesucht, so dass wenn die Stimmen der Parteien in dem Wahlkreis durch den Divisor geteilt werden, gerundet werden, und dann aufaddiert werden, man die Anzahl verfügbarer Sitze in dem Wahlkreis erhält. Beachten Sie, dass nach diesem Schritt die Stimmen Dezimalzahlen sein können (im gegebenen Code nennen wir diese meist "Bruchteile von Sitzen"), wobei der gerundete Wert der Anzahl Sitze für eine Partei in einem Wahlkreis entsprechen wird. Als nächstes wird das Gleiche für jede Partei gemacht, nur dass die Summe danach der Sitze der Partei entsprechen soll. Ein gutes Beispiel finden Sie [hier](#) unter "Unterzuteilung". Da beim Anpassen einer Folge für einen Wahlkreis die Folge für eine

Partei geändert werden kann, wird dieses Verfahren wiederholt, bis die Summe der gerundeten Bruchteile von Sitzen in jeder Folge für jeden Wahlkreis und jede Partei der Anzahl verfügbarer Sitze entspricht. Ähnlich wie die Zuteilungsstrategie, wird die Untertzuteilung parametrisiert mit einer Rundungsstrategie, welche vorgibt wie gerundet werden soll. Definieren Sie dafür die nötigen Methoden in dem Interface `RundungStrategy` und implementieren Sie die `getter` in der `RundungStrategyFactory` Klasse. Weitere Anweisungen stehen im Code.

Mini-Taschenrechner (2018 W10)

In der letzten Bonusaufgabe sollten Sie zeigen, dass Sie mit der Theorie von Klassen und Vererbung umgehen können. In dieser Aufgabe geht es um die praktische Anwendung von Klassen und Vererbung. Sie werden einen Mini-Taschenrechner implementieren der mathematische Ausdrücke (*expressions*) als String einliest, als Baum aufbaut und auswertet. Wir beschreiben die Aufgabe in zwei Teilen.

- Für diese Aufgabe bestehen mathematische Ausdrücke aus Zahlen und den Operatoren $+$, $*$ und einem unären $-$, heisst, ein $-$ das nur ein Argument nimmt. Die genaue Syntax dieser mathematischen Ausdrücke beschreiben wir im zweiten Teil.

Im Programm werden mathematische Ausdrücke als Bäume von Objekten unterschiedlicher Klassen ausgedrückt. Im Anhang 42.1 befinden sich die Interfaces, welche von den Klassen implementiert werden sollen. Die Interfaces sind auch in der Datei `Terme.java` enthalten. Ein Teil der Aufgabe ist es Klassen zu schreiben, welche diese Interfaces implementieren und alle von uns definierten mathematischen Ausdrücke darstellen können. Im Folgenden geben wir eine Beschreibung aller Interfaces:

Expr ist das Super-Interface aller Klassen, welche mathematische Ausdrücke darstellen. Das Interface hat zwei Methoden:

`children()` gibt alle direkten Kinder von einem Ausdruck zurück. Für eine Operation sind das immer alle Operanden und für eine Zahl ist das ein leeres Array.

`eval()` ist das Kernstück unseres Taschenrechners. Es gibt den Integer-Wert eines ausgewerteten Ausdrucks zurück. Zum Beispiel für die Addition von zwei Zahlen gibt es die Summe dieser beiden Zahlen zurück.

BinOp stellt alle binären Operationen dar (Addition und Multiplikation). Die Methoden `left()` und `right()` geben den linken respektive rechten Operanden zurück.

UnaryOp stellt alle unären Operationen dar (Minus). Die Methode `operand()` gibt das einzelne Argument der unären Operation zurück.

Constant stellt alle Zahlen dar. Die Methode `val()` gibt den Wert der Zahl zurück.

- Bevor Sie einen Ausdruck auswerten können, muss zuerst der Baum aufgebaut werden. Implementieren Sie dafür in der Datei `Rechner.java` die Methode `parse`. Die Methode nimmt als Argument einen Scanner und gibt eine Instanz mit Typ `Expr` zurück.

Für diese Aufgabe repräsentieren wir mathematische Ausdrücke in einer Prefix-Notation, heisst, es kommt zuerst der Operator und dann die Operanden. Abbildung 7 zeigt die genaue Syntax dafür. Es werden keine Klammern benötigt, da für unsere Operatoren die Anzahl

$$\begin{aligned}
 \textit{digit} &\Leftarrow 0 \mid 1 \mid \dots \mid 9 \\
 \textit{num} &\Leftarrow \textit{digit} \{ \textit{digit} \} \\
 \textit{plus} &\Leftarrow + \textit{expr} \textit{expr} \\
 \textit{mult} &\Leftarrow * \textit{expr} \textit{expr} \\
 \textit{minus} &\Leftarrow - \textit{expr} \\
 \textit{expr} &\Leftarrow \textit{num} \mid \textit{plus} \mid \textit{mult} \mid \textit{minus}
 \end{aligned}$$

Abbildung 7: EBNF-Beschreibung von *expr*

Operanden immer eindeutig ist. Zum Beispiel wird der Ausdruck $(1+33)*-(2+4)$ dargestellt als `*+133-+24`.

Sie dürfen annehmen, dass nur gültige mathematische Ausdrücke an `parse()` übergeben werden. Alle Operationen und Zahlen sind mit Leerzeichen getrennt. Praktischerweise gibt so die `next()`-Methode des Scanners immer genau einen Operator oder eine Zahl als `String` zurück, solange der Ausdruck nicht komplett gelesen wurde.

Komplementär zu der `parse`-Methode sollen Sie die `toString`-Methode aller Klassen, welche mathematische Ausdrücke repräsentieren, so überschreiben, dass für jede gültige `String`-Repräsentation `str` Folgendes gilt:

```
parse(str).toString().equals( parse( parse(str).toString() ).toString() )
```

In anderen Worten, der zurückgegebene `String` der `toString`-Methode kann wieder zum ursprünglichen Ausdruck aufgebaut werden. Beachten Sie, dass hier `parse` einen `String` als Argument nimmt. Diese Methode ist in der Datei "Rechner.java" durch Aufruf der anderen `parse`-Methode implementiert.

In der Datei "RechnerTest.java" finden Sie ein paar Tests. Sehen Sie sich diese Tests an, falls Teile der Aufgabe unklar sind. Hier eine Zusammenfassung der einzelnen Aufgaben:

- Schreiben Sie Klassen, welche alle arithmetischen Ausdrücke (siehe Abbildung 7) darstellen können und welche die Interfaces (siehe Anhang 42.1) korrekt implementieren.
- Implementieren Sie die `parse`-Methode der Datei "Rechner.java", welche als Argument einen `Scanner` nimmt und eine `Expr` zurückgibt.
- Überschreiben Sie die `toString`-Methode aller Klassen, welche `Expr` implementieren, sodass für jede gültige `String`-Repräsentation `str` Folgendes gilt:

```
parse(str).toString().equals( parse( parse(str).toString() ).toString() )
```

Wichtig: Teilpunkte können nur erreicht werden, wenn `parse` implementiert ist.

Tipp: Implementieren Sie `parse` rekursiv.

Testprogramm Bonusaufgabe

```
interface Expr {
    Expr[] children();
    int eval();
}

interface BinOp extends Expr {
    Expr left();
    Expr right();
}

interface UnaryOp extends Expr {
    Expr operand();
}

interface Constant extends Expr {
    int val();
}
```

Week 11

Contact Tracing (2023 W11)

In dieser Aufgabe implementieren Sie eine Contact-Tracing-Applikation, welche es ermöglichen soll, Kontakte während eines Virus-Ausbruches nachzuverfolgen. Ihre Implementierung soll zunächst Begegnungen zwischen verschiedenen Person-Instanzen anonym protokollieren, so dass bei einem positivem Test die Benachrichtigung aller Personen möglich ist, die direkt oder indirekt mit einer positiv getesteten Person in Kontakt standen.

Anonyme Begegnungen. Um Anonymität zu gewährleisten, dürfen zwei Personen *A* und *B* bei einer Begegnung lediglich anonyme Integer-IDs austauschen, ohne dabei die Identität der jeweils anderen Person aufzudecken. Beide Personen speichern hierbei sowohl die eigene ID als auch die ID der anderen Person. Bei der positiven Testung von *A* kann dann mithilfe der anonymen IDs, die *A* genutzt hat, festgestellt werden, ob *B* einer dieser IDs begegnet ist. Um zu vermeiden, dass wiederkehrende IDs die Identifikation einer Person über mehrere Begegnungen hinweg ermöglichen, benutzt jede Person für jede Begegnung frische IDs, welche über eine zentrale Klasse `ContactTracer` vergeben werden. Frisch bedeutet hierbei, dass eine ID zuvor noch nie bei einer Begegnung verwendet wurde.

Direkte und indirekte Kontakte. Nachdem eine Reihe an Begegnungen protokolliert wurden, wird eine oder mehrere Personen positiv getestet. Mit dem erfassten Netzwerk aus Begegnungen soll Ihre Applikation dann zwei verschiedene Arten an Kontaktpersonen bestimmen:

- Als *direkte Kontakte* gelten alle Personen, die eine Begegnung mit einer positiv getesteten Person hatten.

- Als *indirekte Kontakte* hingegen gelten alle Personen, die zwar selbst keine Begegnung mit einer positiv getesteten Person hatten, jedoch Kontakt mit mindestens einer anderen Person, welche als direkter Kontakt gilt, hatten. Indirekte Kontakte mit mehr als einer Zwischenperson müssen Sie dabei nicht berücksichtigen.

Sie dürfen dabei annehmen, dass zunächst alle Begegnungen erfasst werden und erst dann Personen positiv getestet werden. Nach der ersten positiven Testung finden keine weiteren Begegnungen mehr statt.

Benachrichtigungen. Da nicht alle Personen gleichermassen gefährdet sind, soll Ihre Applikation die Benachrichtigung der Kontaktpersonen vom Alter, der Art des Kontaktes, sowie dem Testergebnis der jeweiligen Kontaktperson abhängig machen. Dabei soll eine der drei Warnstufen *Keine Benachrichtigung*, *Low-Risk-Benachrichtigung* oder *High-Risk-Benachrichtigung* ausgesprochen werden. Zu Beginn haben alle Personen die Standard-Warnstufe *Keine Benachrichtigung* und gelten als negativ getestet. Davon ausgehend sollen nach jedem registrierten positiven Test die zugehörigen Kontaktpersonen wie folgt benachrichtigen werden:

Testergebnis der Kontaktperson	Alter der Kontaktperson	Direkter Kontakt	Indirekter Kontakt
Positiv	-	Keine Benachr.	Keine Benachr.
Negativ	≤ 60 Jahre alt	High-Risk	Keine Benachr.
Negativ	> 60 Jahre alt	High-Risk	Low-Risk

Eine negativ getestete Person, die höchstens 60 Jahre alt ist und die nur in indirektem Kontakt zu einer positiven Person stand, soll beispielsweise keine Benachrichtigung erhalten (Reihe 2). Eine negativ getestete Person über 60 Jahre hingegen soll als indirekter Kontakt eine Low-Risk-Benachrichtigung erhalten (Reihe 3).

Wenn mehrere Personen positiv getestet werden, soll Ihre Applikation immer die höchste geltende Warnstufe für die anderen, negativ getesteten Personen berechnen. Dabei ist die Ordnung der Warnstufen wie folgt definiert: *Keine Benachrichtigung* < *Low-Risk Benachrichtigung* < *High-Risk Benachrichtigung*. Positiv getestete Personen hingegen sollen immer die Warnstufe *Keine Benachrichtigung* erhalten. Im Allgemeinen dürfen Sie zudem annehmen, dass eine Person, die einmal positiv getestet wurde, für den Rest der Laufzeit Ihrer Applikation als positiv getestet gilt.

Implementierung. Erweitern Sie den vorgegebenen Code für die Klasse `ContactTracer` und das Interface `Person` wie folgt, um die Contact-Tracing-Applikation umzusetzen:

Implementieren Sie das Interface `Person` mit den folgenden public Methoden:

- `Person.getUsedIds()`. Diese Methode gibt die Liste aller IDs zurück (`List<Integer>`), die für diese Person als frische ID verwendet wurden, um eine Begegnung zu protokollieren. Nach Hinzufügen einer ID in diese Liste muss dieselbe ID in die jeweilige `Person.getSeenIds()`-Liste des Gegenübers eingetragen sein.
- `Person.getSeenIds()`. Diese Methode gibt die Liste aller IDs zurück (`List<Integer>`), die diese Person als die frische ID des jeweiligen Gegenübers bei einer Begegnung protokolliert hat. Nach Hinzufügen einer ID in diese Liste muss dieselbe ID in die jeweilige `Person.getUsedIds()`-Liste des Gegenübers eingetragen sein.

- `Person.getNotification()`. Diese Methode gibt den aktuellen Benachrichtigungsstatus der Person zurück. Der Rückgabewert soll vom Enum-Typ `NotificationType` sein, welcher vorgegeben ist und die drei möglichen Warnstufen modelliert. `NotificationType` ist im Interface `Person` definiert und enthält die drei Werte `NoNotification` (keine Benachrichtigung), `LowRiskNotification` (Low-Risk-Benachrichtigung) und `HighRiskNotification` (High-Risk-Benachrichtigung).
- `Person.setTestsPositively()`. Diese Methode wird aufgerufen, um eine Person als positiv getestet zu markieren. Nach dem Aufrufen dieser Methode sollen automatisch alle Kontakte von `A` benachrichtigt worden sein und die entsprechenden Warnstufe per `Person.getNotification()` zurückgeben.

Implementieren Sie zusätzlich die Klasse `ContactTracer`, welche die folgenden public Methoden besitzt:

- `ContactTracer.registerEncounter(Person p1, Person p2)`. Mit dieser Methode wird eine (beidseitige) Begegnung zwischen Person-Objekten `p1` und `p2` protokolliert, indem die beiden Personen anonyme IDs austauschen. Die ausgetauschten IDs müssen dabei unterschiedlich sein. Eine Begegnung zwischen `p1` und `p2` ist beidseitig und muss somit auch als Begegnung zwischen `p2` und `p1` gewertet werden.
- `ContactTracer.createPerson(int age)`. Diese Methode gibt ein Person-Objekt zurück. Das Alter der Person ist durch den `age` Parameter bestimmt.

Alle Person-Objekte werden von der Methode `ContactTracer.createPerson(int age)` erstellt. Der `ContactTracer` wird über den parameterfreien Konstruktor `ContactTracer()` instanziiert. Sie dürfen annehmen, dass nie mehr als 1024 Begegnungen zwischen Personen protokolliert werden.

Implementieren Sie auf Basis dieser Vorlage eine Lösung für das Contact-Tracing-Problem. Tests finden Sie in der Datei `“ContactTracerTest.java”`. Die Datei `“ContactTracerGradingTest.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.

Biome (2022 W11)

In dieser Aufgabe implementieren Sie zwei morphologische Operationen auf einem $n \times n$ Grid von *Biomen* (Arten von Landschaften), wobei immer $n \geq 2$ gilt. Es gibt zwei Arten von Biomen: Flachland und Wasser. Jedes Biom hat eine Floradiversität, welches ein Mass für die Pflanzendiversität ist und durch eine nicht-negative ganze Zahl (≥ 0) gemessen wird. Zusätzlich haben Flachlandbiome eine Höhe, welche als eine strikt positive ganze Zahl (> 0) angegeben wird.

Im vorgegeben Programmskelett ist eine Klasse `World` gegeben und ein Interface `Biom`. Die Klasse `World` soll eine Repräsentation eines $n \times n$ Grid von Biomen speichern und darauf die morphologischen Operationen `World.stepDryUp()` und `World.stepDistribute(int p)` ausführen. Das Interface `Biom` repräsentiert Biome und schreibt drei Methoden vor: `Biom.getFlora()` (soll die Floradiversität zurückgeben), `Biom.getHeight()` (soll die Höhe bei Flachlandbiomen zurückgeben; beim Wasser muss immer 0 zurückgegeben werden), und `Biom.getBiomType()`,

welche die Biomart als String zurückgibt. Die String-Repräsentation der Biomarten sind durch folgende Strings der Länge 1 definiert:

Biomart	Wasser	Flachland
<code>Biom.getBiomType()</code>	"W"	"F"

Konkrete Implementierungen von Biom für die unterschiedlichen Biomarten sind nicht gegeben und diese müssen Sie in den Unteraufgaben selber implementieren. Sie dürfen weitere Methoden zum Biom-Interface oder zur Klasse World hinzufügen (aber keine Methoden entfernen oder die Interfaces durch Klassen ersetzen). Der Entwurf der Klasse(n) und gegebenenfalls weiterer Interfaces ist Ihnen überlassen.

- a) Vervollständigen Sie den Konstruktor `World(String[] [] biomGrid)`, welcher als Input ein 2D-Array `biomGrid` nimmt. `biomGrid` gibt das $n \times n$ Startgrid von Biomen für das World-Objekt vor ($n = \text{biomGrid.length}$ und $n \geq 2$). `biomGrid[i][j]` gibt die String-Repräsentation der Biomart an Stelle (i, j) an ($0 \leq i, j < n$). Die Floradiversität der Biome soll am Anfang für jedes Wasserbiom mit 15 und für jedes Flachlandbiom mit 12 initialisiert werden. Die Höhe der Flachlandbiome soll mit 3 initialisiert werden. Implementieren Sie zusätzlich die Methode `World.getBiom(int x, int y)`, welche das Biom an Stelle (x, y) vom Grid als Objekt vom Typ Biom zurückgibt.

Im nächsten Teil der Aufgabe geht es darum morphologische Operationen `stepDryUp()` und `stepDistribute(int p)` in World zu implementieren, welche das Grid von World potenziell ändern (die Grösse des Grids bleibt jedoch gleich). Morphologische Operationen wandeln ein Grid G in ein Grid G' um. Sei $G_{i,j}$ (bzw. $G'_{i,j}$) das Biom an Stelle (i, j) im Grid G (bzw. G'). Dabei wird das neue Biom $G'_{i,j}$ an Stelle (i, j) immer nur abhängig vom alten Biom an Stelle (i, j) (also $G_{i,j}$) und den alten Nachbar-Biomen von $G_{i,j}$ berechnet. Für `stepDryUp()` spielen die Nachbar-Biome keine Rolle. Für `stepDistribute(int p)` sind die Nachbar-Biome von $G_{i,j}$ die Biome $G_{i,j-k}$, $G_{i,j+k}$, bzw. $G_{i-k,j}$ und $G_{i+k,j}$ für alle k mit $0 < k \leq p$. Das heisst, für $p = 1$ sind die Nachbar-Biome gegeben durch $G_{i,j-1}$ und $G_{i,j+1}$ [links und rechts daneben] und die Biome $G_{i-1,j}$ und $G_{i+1,j}$ [oben und unten]. Sie dürfen davon ausgehen, dass $p \in \{1, 2\}$ gilt.

Nur die Elemente des Grids, die wirklich existieren, spielen eine Rolle. Zum Beispiel sind für `stepDistribute(1)` die Nachbar-Biome von $G_{0,0}$ immer nur $G_{0,1}$ und $G_{1,0}$ (da die Positionen $(0, -1)$, $(-1, 0)$ nicht existieren). Für `stepDistribute(2)` bei einem $n \times n$ Grid mit $n \geq 3$ sind die Nachbar-Biome von $G_{0,0}$ immer nur $G_{0,1}$, $G_{1,0}$, $G_{0,2}$, $G_{2,0}$. Bei `stepDistribute(int p)` hat ein Biom maximal $4p$ Nachbar-Biome.

Tabelle 1 (unten) beschreibt für Wasserbiome wie man für die beiden Operationen die neuen Biome von den alten Biomen berechnet; Tabelle 2 (unten) beschreibt dies für Flachlandbiome. Bei `stepDryUp` berechnet man die neue Floradiversität eines Bioms ($\text{Diversität}_{\text{neu}}$) aus der alten Floradiversität des Bioms an der gleichen Position. Bei `stepDistribute` berechnet man die neue Floradiversität eines Bioms aus den alten Floradiversitäten der Nachbarbiome. Die neue Höhe (Höhe_{neu}) von Flachlandbiomen ist bei beiden Operation von der alten Höhe des Bioms an der gleichen Position abhängig (wobei bei `stepDryUp` ein Flachlandbiom in ein Wasserbiom umgewandelt werden kann).

Lösen Sie die Aufgaben b) und c).

- b) Implementieren Sie die Methode `World.stepDryUp()` (siehe Tabellen).

Beispiel: Angenommen ein Flachlandbiom hat vorher Floradiversität 5 und Höhe h . Wenn $h - 1 \leq 0$, dann wandelt sich das Biom in ein Wasserbiom mit Diversität $5 - 3 = 2$. Wenn $h - 1 > 0$, dann bleibt das Biom ein Flachlandbiom, aber mit Diversität $5 - 3 = 2$ und Höhe $h - 1$.

- c) Implementieren Sie die Methode `World.stepDistribute(int p)` (siehe Tabellen), wobei Sie annehmen dürfen, dass $p \in \{1, 2\}$.

Beispiel: Angenommen ein Flachlandbiom an Stelle (i, j) hat vorher Höhe 8 und zusätzlich hat das Flachlandbiom zwei Flachlandnachbar-Biome und zwei Wassernachbar-Biome. Die Flachlandnachbarn haben Diversität 4 bzw. 5 und die Wassernachbarn haben Diversität 6 bzw. 7. Dann bleibt das Biom an Stelle (i, j) ein Flachlandbiom, aber mit Diversität $4 + 5 + 6 + 7 = 22$ und Höhe $8 + 2 = 10$.

Tests finden Sie in der Datei "BiomTest.java", welche auch an der Prüfung zur Verfügung gegeben wurde, und "GradingBiomTest.java", welche zur Korrektur der Prüfung verwendet wurde. Versuchen Sie zuerst die Tests aus "BiomTest.java" zu bestehen und prüfen Sie dann, ob Sie die Tests aus "GradingBiomTest.java" bestehen. Beachten Sie, dass die "BiomTest.java" die Methode `World.stepDistribute(int p)` nur für $p = 1$ testet. Für die volle Punktzahl muss Ihre Implementierung auch für $p = 2$ funktionieren.

Methode	Floradiversität
<code>stepDryUp</code>	$\text{Diversität}_{\text{neu}} = \text{Max}(\text{Diversität}_{\text{alt}} - 5, 0)$
<code>stepDistribute</code>	$\text{Diversität}_{\text{neu}} = \text{Summe der Diversität}_{\text{alt}}$ aller Nachbar-Biome

Tabelle 1: Operationen für Wasserbiome. Ein Wasserbiom bleibt nach einer Operation immer ein Wasserbiom.

Methode	Floradiversität	Höhe
<code>stepDryUp</code>	$\text{Diversität}_{\text{neu}} = \text{Max}(\text{Diversität}_{\text{alt}} - 3, 0)$	$\text{Höhe}_{\text{neu}} = \text{Höhe}_{\text{alt}} - 1$ Wenn $\text{Höhe}_{\text{neu}} = 0$ dann verwandelt sich das Biom in ein Wasser Biom (mit $\text{Diversität}_{\text{neu}}$)
<code>stepDistribute</code>	$\text{Diversität}_{\text{neu}} = \text{Summe der Diversität}_{\text{alt}}$ aller Nachbar-Biome	$\text{Höhe}_{\text{neu}} = \text{Höhe}_{\text{alt}} + (\text{Anzahl Nachbar-Biome die Flachland sind})$

Tabelle 2: Operationen für Flachlandbiome. Nach einer `stepDryUp` Operation wandelt sich ein Flachlandbiom in ein Wasserbiom, wenn die alte Höhe 1 ist. In allen anderen Fällen bleibt ein Flachlandbiom nach einer Operation ein Flachlandbiom.

Hamster Kartell (2021 W11)

Durch eine Indiskretion sind Ihnen vertrauliche Daten des Hamsterzüchterverbandes zugespielt worden. In einer Datei finden Sie Informationen über alle angebotenen Tiere des Verbandes. Für jedes Angebot gibt es einen Eintrag auf einer Zeile in der Datei. Ein Eintrag auf einer Zeile enthält folgende Informationen (in dieser Reihenfolge):

- Der angebotene Preis vom Tier (ein positiver Integer).
- Die Gattung vom Tier (ein String).
- Das Alter vom Tier (ein positiver Integer).
- Die ID vom Tier (ein positiver Integer). Sie können annehmen, dass jedes Tier eine einzigartige ID hat.
- Der Name des Züchterverbandes (ein String).

In dieser Aufgabe werden Sie verschiedene Methoden implementieren, welche die Einträge der Datei analysieren.

1. Implementieren Sie den Konstruktor `HamsterAnalysis(Scanner input)`. Die Methode nimmt als Argument einen Scanner von dem Sie den Inhalt der Eingabe-Datei lesen sollen. Es müssen nur korrekt formatierte Eingabe-Dateien unterstützt werden. Ein Beispiel einer solchen Datei finden Sie in der Datei "hamster.txt". Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren. Implementieren Sie zusätzlich die Methode `HamsterAnalyse.getData()`, welche eine neue Liste mit den Angeboten aus der Eingabe-Datei zurückgibt (in der gleichen Reihenfolge). Die Klasse `Offer` repräsentiert ein Angebot und bietet die entsprechenden Attribute zur Verfügung.
2. Implementieren Sie die Methode `HamsterAnalysis.genus(String gen)`. Die Methode gibt eine Liste von Integer zurück. Die zurückgegebene Liste soll die IDs von den Angeboten der Eingabe-Datei enthalten, welche als Gattung das Argument `gen` haben. Die zurückgegebenen IDs sollen die gleiche Reihenfolge haben wie in in der Eingabe-Datei.
3. Implementieren Sie die Methode `HamsterAnalysis.fraudDistance(String gen, int age)`. Die Methode gibt einen Integer n zurück. Der Integer n ist die grösste Differenz zwischen dem Preis von einem Angebot, welches als Gattung `gen` und als Alter `age` hat, und dem (zu einem `int` aufgerundeten) durchschnittlichen Preis aller Angebote, welche als Gattung `gen` und als Alter `age` haben. Als Beispiel, wenn die Preise der Angebote für die entsprechende Gattung und das entsprechende Alter gleich `[3,3,4,4,4,6]` sind, dann soll 2 zurückgegeben werden. Falls es kein Angebot gibt, welches als Gattung `gen` und als Alter `age` hat, dann soll `-1` zurückgegeben werden.

In der Datei "HamsterAnalysisTest.java" finden Sie ein paar Beispiele und Tests.

Hogwarts (2020 W11)

In dieser Aufgabe implementieren Sie das Punktesystem von Hogwarts, bei welchem Studenten eines Hauses Punkte verliehen oder abgezogen bekommen können und dadurch der kumulative Punktestand ihres Hauses sich verändert. Wir verwenden drei Klassen, `School`, `House`, und `Student`, für die Schule, Häuser, und Studenten. Die Klassen können folgendermassen verwendet werden:

```
School hogwarts = new School();

// Haeuser werden erstellt (Anzahl Haeuser und Namen sind nicht eingeschaenkt).
House hufflepuff = hogwarts.createHouse("Hufflepuff");
House ravenclaw = hogwarts.createHouse("Ravenclaw");

// Studenten haben Vor- und Nachnamen.
Student hannah = new Student("Hannah", "Abbott");
Student newton = new Student("Newton", "Scamander");
Student luna = new Student("Luna", "Lovegood");
Student filius = new Student("Filius", "Flitwick");

// Studenten werden den Haeusern zugeordnet.
hufflepuff.assign(hannah);
hufflepuff.assign(newton);
ravenclaw.assign(luna);
ravenclaw.assign(filius);

// Punkte werden an Studenten vergeben. Punkte koennen auch negativ sein.
hannah.givePoints(10);
newton.givePoints(-5);
luna.givePoints(8);

// Informationen zu der Summe an Punkten und dem aktuellen Siegerhaus
// koennen immer abgefragt werden.
System.out.println("Siegerhaus:␣" + hogwarts.winner().name());
System.out.println("Siegerpunkte:␣" + hogwarts.winner().points());
System.out.println("Hogwarts␣Punkte␣Insgesamt:␣" + hogwarts.points());
```

In der Vorlage finden Sie das Skelett für die drei Klassen, eine Klasse `SchoolExample` mit dem obigen Beispiel, sowie eine Test-Klasse `SchoolTest`, welche Sie als Starthilfe für das Testen Ihrer Lösung verwenden können. Zusätzlich gibt es eine Test-Klasse `SchoolTestExam`. Diese enthält Tests, welche an einer Prüfung zur Korrektur verwendet werden könnten. Wir empfehlen stärkstens, dass Sie diese Tests erst verwenden, wenn Sie davon überzeugt sind, dass Sie die Aufgabe korrekt gelöst haben.

1. Implementieren Sie den `School`-Konstruktor und die Methode `createHouse(String name)`, welche als Parameter den gewünschten Namen des Hauses nimmt und ein `House` Objekt zurück zurückgibt. Der Name eines Hauses darf nicht null sein oder bereits für die Schule vorhanden sein. Die Methode soll in diesen Fällen eine `IllegalArgumentException` werfen. Alle anderen Namen sind erlaubt. Implementieren Sie zusätzlich die Methode `name()` der Klasse `House`, welche den Namen des Hauses als `String` zurückgibt.

2. Implementieren Sie den Konstruktor von `Student`, welcher zwei Strings, den Vor- und Nachnamen (in dieser Reihenfolge) nimmt. Sie dürfen annehmen, dass es jeden Namen (Vor- und Nachname zusammen) nur einmal gibt. Vor- und Nachnamen sollen über die Methode `firstName()` beziehungsweise `lastName()` erhalten werden können. Implementieren Sie zusätzlich die Methode `assign(Student student)` der Klasse `House`, welche einen Studenten als Argument nimmt und ihn in dieses Haus einschreibt. Bei einem `null` Argument oder falls der Student bereits bei einem Haus der gleichen Schule eingeschrieben ist, dann soll die Methode eine `IllegalArgumentException` werfen.
3. Als letztes implementieren Sie das Punktesystem. Implementieren Sie dafür vier Methoden: Die Methode `points()` von `House` gibt die Punkte eines Hauses zurück. Jedes Haus beginnt mit einem Punktestand von 0, wenn es erstellt wird. Dieser Punktestand kann sich dann durch die Leistungen der Studenten verändern. Die Methode `givePoints(int points)` von `Student` nimmt eine positive oder negative Anzahl Punkte, welche dem Studenten verliehen werden. Erhaltene Punkte zählen nur, wenn der Student einem Haus bereits zugewiesen wurde. Die erhaltenen Punkte werden dann den Häusern zugeschrieben, welchen der Student zugewiesen ist. Dabei können die Punkte eines Hauses nicht kleiner als 0 werden. Auch wenn einem Studenten mehr Punkte abgezogen werden, geht der Punktestand eines Hauses nur auf 0. Zum Beispiel, wenn Hufflepuff in der Summe 5 Punkte hat und Hannah -10 Punkte verliehen werden, dann werden nur -5 Punkte tatsächlich für Hufflepuff verrechnet, der Rest wird ignoriert. Zusätzlich implementieren Sie die Methode `winner()` von `School`, welche das Haus mit den meisten Punkten zurückgibt. Falls mehrere Häuser die gleiche Punktzahl haben, dann kann ein beliebiges dieser Häuser zurückgegeben werden. Falls es kein Haus gibt, dann soll die Methode eine `IllegalArgumentException` werfen. Und implementieren Sie die Methode `points()` von `School`, welche die Summe der Punktestände der Häuser zurückgibt.

Notenauswertung (2019 W11)

Die Klasse `Service` stellt verschiedene Analysen für Prüfungsergebnisse von S Studierenden zur Verfügung. Die Liste von Ergebnissen besteht aus S Einträgen, also jeweils ein Eintrag pro Student/in. Jeder Eintrag besteht aus einer Zeile und enthält (in dieser Reihenfolge):

1. die Immatrikulationsnummer des Studierenden (ein identifizierender positiver `int`-Wert)
2. drei Noten (drei reelle Zahlen im Bereich von 1.0 bis 6.0, getrennt durch Leerzeichen)

Die drei Noten gehören zu den Fächern *Fach 1*, *Fach 2* und *Fach 3*. Zusätzliche Leerzeilen und -zeichen sollen ignoriert werden. Eine Beispiel für eine Liste für 3 Studierende ist:

```
111111004  5.0  5.0  6.0
111111005  3.75 3.0  4.0
111111006  4.5  2.25 4.0
```

Ihre Aufgabe ist es nun, die `Service`-Klasse und ihre Analysen zu implementieren. Die `Service`-Klasse hat einen Konstruktor, welcher alle Prüfungsergebnisse aus einem Scanner auslesen und damit das `Service`-Objekt initialisieren soll. Das Objekt soll so initialisiert werden, dass

die vorgegebenen Methoden ihre Analysen durchführen können. Sie dürfen dabei Attribute und zusätzliche Methoden frei bestimmen.

- a) Implementieren Sie nun die Methode `critical()`, welche die zwei Argumente `bound1` und `bound2` erwartet. Die Methode sucht alle "kritischen" Fälle und gibt eine Liste dieser Studierenden zurück. Ein Student darf maximal einmal in der Liste vorkommen. Die zurückgegebene Liste besteht aus den Immatrikulationsnummern dieser Studierenden (in beliebiger Reihenfolge).

Ein/e Student/in gilt als kritisch, wenn die Note in *Fach 1* \leq `bound1` und die Summe der Noten für *Fach 2* und *Fach 3* kleiner als `bound2` ist.

Für das obige Beispiel gäbe `critical(4, 8)` eine Liste mit dem Element `111111005` zurück.

- b) Implementieren Sie nun die Methode `top()`, welche die Studierenden mit den besten Ergebnissen zurückgeben soll. Der Parameter `limit` bestimmt die maximale Anzahl der zurückzugebenden Studierenden. Falls weniger Ergebnisse als `limit` existieren, sollen einfach alle gefundenen zurückgegeben werden.

Der Rückgabewert der Methode ist wieder eine Liste der Immatrikulationsnummern. Ein Student darf maximal einmal in der Liste vorkommen. Diese Liste soll absteigend nach der Leistung sortiert sein (der/die Student/in mit dem besten Ergebnis zuerst). Dabei gilt, dass ein Ergebnis *A* besser ist als ein Ergebnis *B*, wenn die Summe aller Noten von *A* grösser ist als die Summe der Noten von *B*. Sind die Summen gleich, sind die Ergebnisse gleich gut (und die Reihenfolge in der Liste somit egal).

Für das obige Beispiel gäbe `top(2)` entweder die Liste `[111111004, 111111006]` oder die Liste `[111111004, 111111005]` zurück (beide wären richtig).

In der Klasse `ServiceTest` finden Sie einen ersten kleinen JUnit-Test als Starthilfe. Ausserdem dürfen Sie folgende Annahmen machen: Der Parameter `limit` ist immer grösser als 0 und die beiden Parameter für `critical()` sind immer im Bereich von 0.0 bis 100.0.

Tipp: Verwenden Sie die `Collections.sort(...)` Funktion einer Kollektion, welche mit `import java.util.Collections;` importiert werden kann. Beachten Sie, dass dafür die Klasse, welche Sie für die Elemente der Kollektion verwenden, das Interface `Comparable<T>` (T sollte die Klasse selber sein), und damit auch eine Funktion `compareTo` implementieren muss. Diese Funktion nimmt eine Instanz der selben Klasse und gibt 0 zurück, wenn `this` und das Argument gleich sind, gibt 1 zurück, wenn `this` grösser als das Argument ist, und gibt -1 zurück, wenn `this` kleiner als das Argument ist.

Bienen Syndikat (2018 W11)

Durch eine Indiskretion sind Ihnen vertrauliche Daten des Bienenzüchterverbandes zugespielt worden. In einer Datei finden Sie Informationen über die Mitglieder des Leitungsausschusses des Verbandes. Für jedes Mitglied gibt es eine Zeile in der Sie finden:

- Den Nachnamen des Mitglieds (ein Wort). Sie können davon ausgehen, dass alle Mitglieder unterschiedlich heissen.

- Das Land, das dieses Mitglied vertritt. Es wird die für Internet-Domainnamen übliche Abkürzung verwendet, also 2 Buchstaben wie "fr" für Frankreich und "uk" für das Vereinigte Königreich.
- Die offiziell gezahlten Sitzungsgelder (in EUR), als positive ganze Zahl.
- Die nicht versteuerten "Sonderzahlungen" (in EUR), als positive ganze Zahl.

Schreiben Sie ein Programm, welches aus diesen Daten folgende Informationen berechnet:

1. Den Namen des Mitglieds, das die höchste Gesamtzahlung (Sitzungsgeld + Sonderzahlung) erhalten hat, und den Betrag.
2. Den Namen des Mitglieds, für welches die Sonderzahlung den grössten Anteil an der Gesamtzahlung ausmachte, und den Prozentsatz.
3. Das Land, dessen Mitglieder die höchste Summe an Sonderzahlungen erhielten, sowie den Betrag.

Falls mehrere Mitglieder oder Länder für einen der Punkte in Frage kommen, kann das Programm einen beliebigen von ihnen ausgeben.

Vervollständigen Sie die `analyze()`-Methode in der Klasse `Bienen`. Die Methode hat zwei Argumente: Einen Scanner von dem Sie den Inhalt der Eingabe-Datei lesen sollen und einen `PrintStream` in welchen Sie die oben genannten Informationen schreiben.

Ihr Programm muss nur korrekt formatierte Eingabe-Dateien unterstützen. Ein Beispiel einer solchen Datei finden Sie im Projekt unter dem Namen "bienen.txt". Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren.

Jede der Informationen 1.–3. soll (in dieser Reihenfolge) auf einer separaten Zeile ausgegeben werden. Alle Zahlen sollen **kaufmännisch gerundet** als ganze Zahlen ausgegeben werden. Trennen Sie Name (bzw. Land) und Zahl jeweils durch ein einzelnes Leerzeichen. In der Datei "BienenTest.java" finden Sie einen einfachen Test um das Format Ihres Outputs zu testen.

Week 12

Datenbanken (2023 W12)

Diese Aufgabe basiert auf dem Bonus von Aufgabenblatt 10 mit einer anderen Unteraufgabe. Änderungen sind in bold markiert. In dieser Aufgabe implementieren Sie für eine Datenbank von Personengesundheitsdaten **das Erheben von Statistiken (Task a)**.

Die Datenbank selber ist bereits mit der Klasse `Database` implementiert. Die Datenbank hält eine Liste von Einträgen, welche durch die Klasse `Item` repräsentiert werden. Die folgenden 4 Paragraphen erklären alle in der Vorlage gegebenen Klassen im Detail.

Item Die Klasse `Item` repräsentiert einen Datenbankeintrag mit 4 Attributen: eine ID (int), ein Alter (int), einen Gesundheitswert (int), und ein Sicherheitslevel, welches durch die Klasse `Level` repräsentiert wird. Alter und Gesundheitswert sind immer ≥ 0 . Die Methoden `Item.getID()`,

`Item.getAge()`, `Item.getHealth()`, `Item.getLevel()` geben jeweils die ID, das Alter, den Gesundheitswert, und das Sicherheitslevel eines Eintrags zurück. Die Methode `Item.setHealth(int newHealth)` setzt den Gesundheitswert auf `newHealth`. Die anderen Attribute können nicht geändert werden.

Level Die Klasse `Level` repräsentiert ein Sicherheitslevel. Ein Sicherheitslevel wird über ein **Set** von Integern definiert, welches in einem Attribut der Klasse `Level` gespeichert wird und von der Methode `Level.getPoints()` zurückgegeben wird. **Ein Level A ist schwächer als ein Level B, falls es für jeden Wert in `A.getPoints()` einen grösseren Wert in `B.getPoints()` gibt. Zum Beispiel sind die Level $\{2,3\}$ und $\{3,4\}$ schwächer als das Level $\{3,5\}$, da 5 grösser als die anderen Werte ist. Die Level $\{5\}$ und $\{4,6\}$ hingegen sind nicht schwächer als $\{3,5\}$, da es keinen Wert grösser als 5 und 6 in $\{3,5\}$ gibt.**

ItemFactory Die Klasse `ItemFactory` wird verwendet, um Datenbankeinträge zu erstellen. Die Methode `ItemFactory.createItem(Level level, int id, int age, int health)` gibt ein Exemplar der Klasse `Item` zurück, deren Attribute mit den Argumenten initialisiert wurden.

Database Die Klasse `Database` repräsentiert eine Datenbank und hat folgende vorgegebene Methoden:

- `Database.getItemFactory()` gibt ein Exemplar von `ItemFactory` zurück. Die `ItemFactory` `I` ist assoziiert mit der Datenbank `D`, falls `I` von `D.getItemFactory()` zurückgegeben wird.
 - `Database.add(Item item)` fügt der Datenbank den Eintrag `item` hinzu.
 - `Database.getItems()` gibt die Liste aller Einträge zurück, welcher der Datenbank hinzugefügt wurden. Sie dürfen annehmen, dass für eine Datenbank `D` alle Einträge in `D.getItems()` eine einzigartige ID haben, über `D.add` hinzugefügt wurden, über `D.getItemFactory()` erstellt wurden, und keiner anderen Datenbank hinzugefügt werden. Ein hinzugefügter Eintrag wird nie wieder entfernt.
- a) Implementieren Sie die Methode `Database.summary(Level groupLevel)`, welche die durchschnittlichen Alter für Gruppen von Einträgen berechnet. Die Gruppen sind wie folgt definiert: Für jede ganze Zahl k , die ein Vielfaches von 10 ist, besteht die Gruppe für k aus der Menge aller Einträge `E`, so dass
- (a) das Level von `E` schwächer ist als das Argument `groupLevel`; *und*
 - (b) der abgerundete Gesundheitswert von `E` gleich k ist (in Java gilt `E.getHealth() / 10 * 10 == k`).

Die Methode gibt eine `Map <Integer, Integer>` zurück, die für jede wie oben definierte Gruppe das durchschnittliche Alter speichert. Für einen Schlüssel k gibt die Map den Wert `null` zurück, falls k kein Vielfaches von 10 ist oder die Gruppe für k leer ist. Die Map gibt für einen Schlüssel k einen Wert v ungleich `null` zurück, genau dann wenn **alle** folgenden Bedingungen erfüllt sind:

- k ist ein Vielfaches von 10;
- Die Gruppe der gefundenen Einträge für k ist nicht leer;
- v ist das runter-gerundete durchschnittliche Alter aller Einträge in der Gruppe.

ID	Level	Alter	Health
1	{1,2}	20	100
2	{2,3}	31	109
3	{5,6}	50	100
15	{4}	20	133

Beispiel: Für die obige Datenbank mit 4 Einträgen soll `summary` für das Level $\{3,5\}$ die Map $\{100 \mapsto 25, 130 \mapsto 20\}$ zurückgeben. Das Level der Einträge mit ID 1, 2 und 15 ist schwächer als $\{3,5\}$. Die beiden Einträge 1 und 2 sind in einer Gruppe für $k = 100$. Das gerundete durchschnittliche Alter ist $51/2 = 25$. Der Eintrag mit ID 15 ist allein in der Gruppe für $k = 130$. Die Map hat keine weiteren Einträge. Der Eintrag mit ID 3 hat keinen Einfluss auf das Ergebnis, weil sein Level nicht schwächer als $\{3,5\}$ ist.

Wir geben zwei Testdateien zur Verfügung. “DatabaseTest.java” enthält Tests, welche wir an einer Prüfung geben würden. “GradingDatabaseTest.java” enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie Ihre Lösung zuerst ausgiebig mit “DatabaseTest.java” (am besten fügen Sie selber neue Tests hinzu) und dann können Sie “GradingDatabaseTest.java” verwenden, um zu sehen wie Ihre Lösung an einer Prüfung abgeschnitten hätte.

Konstruktion (2022 W12)

In dieser Aufgabe implementieren Sie ein Verteilungssystem für Ressourcen eines Bauunternehmens, mit welchem Ressourcen (z.B. Sandsäcke, Kabelrollen, Ziegelpaletten) auf Baustellen verteilt werden. Das Interface `Resource` repräsentiert Ressourcen. Ressourcen haben einen Typ, welchen wir als Integer repräsentieren. Die Methode `Resource.type()` gibt den Typen einer Ressource zurück. Bauunternehmen und Baustellen gehören Ressourcen. Dabei gehört jede einzelne Ressource immer nur maximal entweder einem einzigen Bauunternehmen oder einer einzigen Baustelle. Der Aufgabentext beschreibt wann und wie sich der Besitz von Ressourcen ändert. Gleichermassen gehört jede Baustelle zu einem einzigen Bauunternehmen. Beim Erstellen der Baustelle wird angegeben, welche Ressourcetypen auf der Baustelle verwendet werden und wie viele Ressourcen des gleichen Typs maximal der Baustelle gehören dürfen.

Die Klasse `CCompany` repräsentiert ein Bauunternehmen (construction company) und hat folgende Methoden:

- `CCompany.resources()` gibt ein Set aller Ressourcen zurück, welche dem Bauunternehmen gehören. Zum Zeitpunkt, wenn ein Bauunternehmen erzeugt wird, gehören dem Bauunternehmen noch keine Ressourcen.
- `CCompany.add(Resource resource)` übergibt dem Bauunternehmen die Ressource `resource`, worauf dem Bauunternehmen die Ressource gehört. Sie dürfen annehmen, dass `resource` vor dem Aufruf keinem anderen Bauunternehmen und keiner Baustelle gehört.
- Die drei Methoden `CCompany.createCSite(Set<Integer> types, int limit)`, `CCompany.createCSite(int type)`, und `CCompany.createCSite(Set<Integer> types, int limit, int flowLimit)` erzeugen eine neue Baustelle, welche dem Bauunternehmen gehört, bis die Baustelle geschlossen wird.

- `CCompany.nextDay()` löst das Übergeben der Ressourcen, welche dem Bauunternehmen gehören, an die Baustellen, welche dem Bauunternehmen gehören, aus. Die Unteraufgaben beschreiben, wie Ressourcen an die Baustellen übergeben werden. Eine übergebene Ressource gehört nicht mehr dem Bauunternehmen (sondern der Baustelle).

Das Interface `CSite` repräsentiert eine Baustelle (construction site) und hat 5 Methoden:

- `CSite.resources()` gibt ein Set aller Ressourcen zurück, welche der Baustelle gehören. Zum Zeitpunkt, wenn eine Baustelle erzeugt wird, gehören der Baustelle noch keine Ressourcen.
- `CSite.canAdd(Resource resource)` gibt zurück, ob der Baustelle die Ressource `resource` übergeben werden darf. Die Unteraufgaben beschreiben, wann das der Fall ist.
- `CSite.add(Resource resource)` übergibt der Baustelle die Ressource `resource`, worauf der Baustelle die Ressource gehört. Die Methode wirft eine `IllegalArgumentException`, falls ein Aufruf von `CSite.canAdd(resource)` `false` zurückgibt. Für die Tests dürfen Sie annehmen, dass `resource` vor dem Aufruf keinem anderen Bauunternehmen und keiner Baustelle gehört und dass die Methode nicht mehr aufgerufen wird, nachdem durch `CCompany.nextDay()` für die Baustelle die Übergabe von Ressourcen ausgelöst wurde.
- `CSite.use(Resource resource)` verbraucht die Ressource `resource`, welche der Baustelle gehört. Die Methode wirft eine `IllegalArgumentException`, falls das Argument nicht der Baustelle gehört. Nachdem eine Ressource verbraucht wurde, gehört die Ressource nicht mehr der Baustelle (und die Ressource gehört auch nichts anderem mehr).
- `CSite.close()` schliesst eine Baustelle, das heisst, die Baustelle wird vom Bauunternehmen entfernt. Sie dürfen annehmen, dass keine weiteren Methoden auf einer geschlossenen Baustelle aufgerufen werden. Die Ressourcen, welche der Baustelle gehören, ändern sich nicht.

Sie dürfen annehmen, dass, solange die Aufgabenstellung nichts anderes vorgibt, alle Argumente nie null sind.

Figure 8a zeigt ein Bauunternehmen Q mit 2 Baustellen X und Y, wobei X vor Y erstellt wurde; das Lager vom Bauunternehmen Q enthält 4 Typen von Ressourcen (T_a, T_b, T_c, T_d). Jedes graue Quadrat stellt eine Ressource dar, wobei die Zahlen angeben in welcher Reihenfolge die Ressourcen an das Bauunternehmen Q übergeben wurden. Die grauen Quadrate bei der Baustelle repräsentieren Ressourcen, welche schon der Baustelle gehören. Auf der Baustelle X werden nur Ressourcen vom Typ T_a, T_c und T_d verwendet (bei Baustelle Y sind es T_a, T_b und T_c), wobei pro Typ maximal zwei Ressourcen der Baustelle X gehören dürfen (bei Baustelle Y sind es pro Typ maximal drei).

a) Implementieren Sie die Methoden

- `CCompany.createCSite(Set<Integer> types, int limit)` (Sie dürfen annehmen `limit` ≥ 0). Diese Methode gibt eine Baustelle `site` mit Platz für `limit` Ressourcen jedes Typs zurück. Eine Ressource `resource` darf der Baustelle `site` übergeben werden (das heisst `site.canAdd(resource)` gibt `true` zurück) genau dann, wenn:
 - (a) die Ressource `resource` einen Typ hat, welcher im Set `types` enthalten ist, und

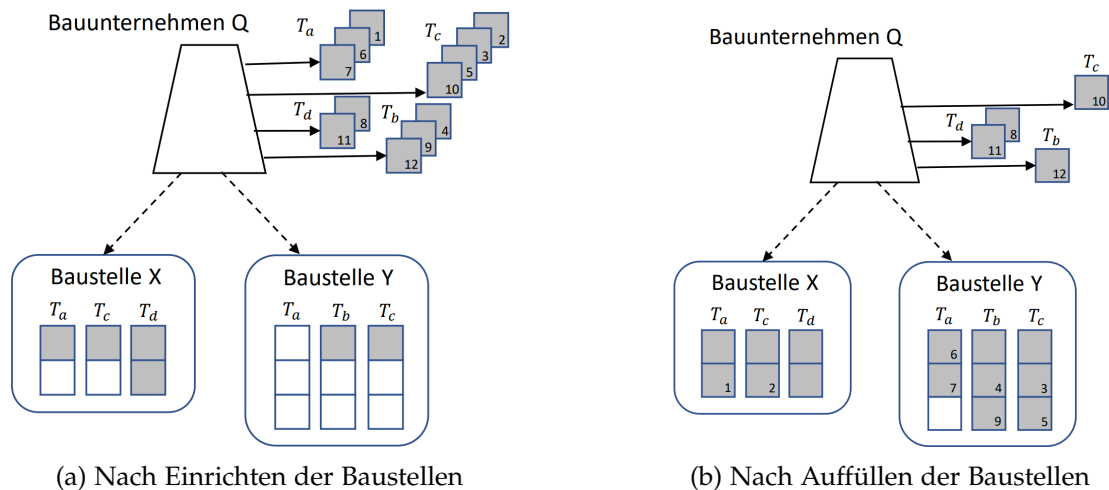


Abbildung 8: Bauunternehmen (Q) mit 2 Baustellen (X und Y) und den 4 Typen (T_a , T_b , T_c , T_d).

(b) falls die Anzahl der Ressourcen, welche der Baustelle gehören und den gleichen Typ wie die Ressource `resource` haben, strikt kleiner als `limit` ist.

- `CSTite.canAdd(Resource resource)`,
- `CSTite.add(Resource resource)`,
- `CSTite.use(Resource resource)`, und
- `CSTite.resources()`.

b) Implementieren Sie `CSTite.close()`, `CCompany.resources()`, `CCompany.add(Resource resource)`, und `CCompany.nextDay()`. Ein Aufruf von `CCompany.nextDay()` sorgt dafür, dass die Ressourcen, welche dem Bauunternehmen gehören, an die Baustellen, welche dem Bauunternehmen gehören, verteilt werden. Ressourcen werden dabei wie folgt an die Baustellen verteilt: Die Ressourcen werden in der Reihenfolge an die Baustellen übergeben, in welcher die Ressourcen an das Bauunternehmen übergeben wurden. In dieser Reihenfolge wird eine Ressource `resource` an die Baustelle `site` übergeben, falls:

- die Ressource `resource` der Baustelle `site` übergeben werden darf, das heisst der Aufruf `site.canAdd(resource)` gibt `true` zurück, und
- falls die Ressource `resource` an keine andere Baustelle `otherSite`, welche dem Bauunternehmen gehört und **vor** der Baustelle `site` erzeugt wurde, übergeben werden darf, das heisst der Aufruf `otherSite.canAdd(resource)` gibt `false` zurück.

Beachten Sie, dass es sein kann, dass eine Ressource an keine Baustelle übergeben wird (diese Ressource verbleibt im Lager und kann vielleicht später übergeben werden).

Figure 8b zeigt den Zustand nach Ausführen der Methode `Q.nextDay()`, wobei nach Einrichten der Baustellen (Figure 8a) und dem Aufruf von `nextDay()` keine Ressourcen hinzugefügt oder verbraucht wurden.

Falls Sie Unteraufgabe (a) nicht gelöst haben, so können Sie Teilpunkte bekommen. Eine Teilmenge der Tests wird nur die Methode `CCompany.createCSTite(int type)` zum Erzeu-

gen von Baustellen verwenden. Diese Methode gibt eine Baustelle zurück, welche eine Ressource nur annehmen darf, falls der Typ der Ressource gleich `type` ist und der Baustelle nicht schon 2 Ressourcen gehören. Um Teilpunkte zu bekommen, können Sie die Methode `CCompany.createCSite(int type)` mit Hilfe der Klasse `TaskBCSite` implementieren, welche bereits alle Methoden der Unteraufgabe (a) implementiert. Beachten Sie, dass Sie die Klasse `TaskBCSite` verändern dürfen.

- c) Implementieren Sie die Methode `CCompany.createCSite(Set<Integer> types, int limit, int flowLimit)` (Sie dürfen annehmen $\text{limit} \geq 0$ and $\text{flowLimit} \geq 0$). Die Methode gibt, wie bei Unteraufgabe (a), eine Baustelle `site` zurück, welcher eine Ressource `resource` nur übergeben werden darf (das heisst `site.canAdd(resource)` gibt `true` zurück) genau dann, wenn:
- i) die Ressource `resource` einen Typ hat, welcher im Set `types` enthalten ist,
 - ii) und falls die Anzahl der Ressourcen, welche der Baustelle gehören und den gleichen Typ wie die Ressource `resource` haben, strikt kleiner als `limit` ist.

Wir nennen die Baustellen, welche von dieser Methode zurückgegeben werden, “dynamische” Baustellen. Eine dynamische Baustelle `site` hat einen “Überfluss” von einem Typ `type`, falls die Anzahl der Ressourcen, welche der Baustelle `site` gehört und Typ `type` hat, strikt grösser als `flowLimit` ist. Dynamische Baustellen erweitern was bei einem Aufruf von `CCompany.nextDay()` passiert: **Bevor die Ressourcen verteilt werden**, übergibt jede dynamische Baustelle `site`, welche dem Bauunternehmen gehört, dem Bauunternehmen die kleinstmögliche Menge an Ressourcen, sodass die Baustelle `site` keinen Überfluss hat. Im Allgemeinen kann es mehrere Mengen geben, welche dieses Kriterium erfüllen und daher gültige Lösungen sind. Eine Ressource, welche dem Bauunternehmen von einer Baustelle übergeben wird, gehört danach dem Bauunternehmen und nicht mehr der Baustelle. Danach werden die Ressourcen verteilt, wie in Unteraufgabe (b) beschrieben.

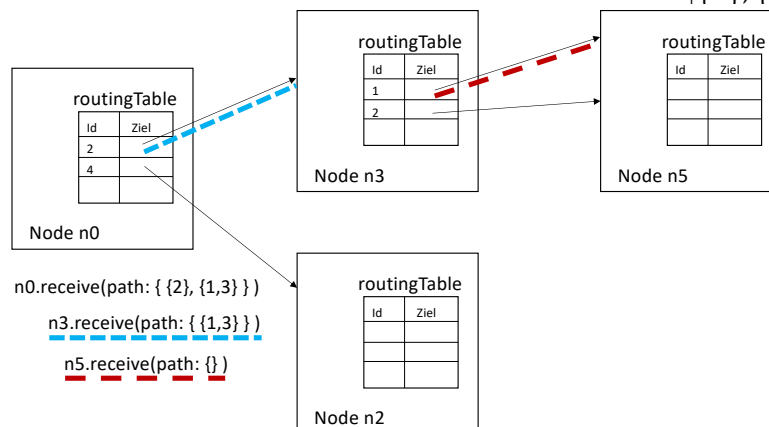
Tests finden Sie in der Datei “`ConstructionCompanyTest.java`”. Die Datei “`GradingConstructionCompanyTest.java`” enthält die Tests, welche wir an der Prüfung für die Korrektur verwendet haben. Wir empfehlen diese Tests erst zu verwenden, wenn Sie denken, dass Ihre Lösung korrekt ist, damit sie sehen können, wie Sie an einer Prüfung abgeschnitten hätten.

Netzwerk (2021 W12)

In dieser Aufgabe implementieren Sie ein Routing Netzwerk, welches Nachrichten durch ein Netz von Knoten leitet. (Sie brauchen keine Netzwerk Kenntnisse, die Aufgabenstellung beschreibt alle Aspekte.) Das Netzwerk besteht aus Knoten, welche das Interface `Node` implementieren. Der erste Teil der Aufgabe ist es, eine Implementation dieses Interfaces (die Klasse `RoutingNode`) zu vervollständigen. Im zweiten Teil werden Sie eine zweite Art von Knoten in einer weiteren Klasse implementieren. Das Interface `Node` erfordert eine Methode `Node.receive(Message msg)`, durch welche ein Knoten `k` die Nachricht `msg` empfängt (also `k.receive(msg)` lässt `k` die Nachricht `msg` empfangen). Was ein Knoten macht, wenn er eine Nachricht empfängt, wird durch die Implementation von `Node` (d.h., der Klasse) bestimmt.

Exemplare der Klasse `RoutingNode` leiten eine empfangene Nachricht an einen Nachbarknoten n weiter (indem sie `n.receive(...)` ausführen). Dazu muss der Knoten k den Zielknoten n finden, an den die Nachricht weitergeleitet wird. Jeder Knoten unterhält Verbindungen zu anderen Knoten. Jede Verbindung besteht aus einer ganzen positiven (`int`) Zahl (der VerbindungsId) und der Referenz des Nachbarknotens, der über diese Verbindung erreichbar ist. Die Verbindungen eines Knotens werden in der Routing Table (dem Attribut `routingTable` der Klasse `RoutingNode`) gespeichert. `routingTable` ist vom Typ `Map<Integer, Node>` (und somit kann es für eine Zahl nur einen Eintrag geben).

Jede Nachricht `msg` enthält den Pfad, den die Nachricht durch das Netzwerk nehmen soll. Ein Pfad ist eine Liste; jedes Element der Liste ist eine Menge von VerbindungsIds, über die eine Nachricht weitergeleitet werden kann. Ein Knoten k benutzt das erste Element der Liste von `msg` um den nächsten Knoten n zu finden und ruft dann die Methode `n.receive(aMsg)` auf. Der Pfad der neuen Nachricht `aMsg` besteht aus dem Pfad der von k empfangenen Nachricht `msg` ohne dem ersten Element (also dem Rest der Liste). Wenn die Liste leer ist, dann hat die Nachricht ihr Ziel erreicht. Die Skizze zeigt den Weg einer Nachricht durch ein (einfaches) Netzwerk (von $n0$ über $n3$ nach $n5$), bei welchem der Knoten $n0$ eine Nachricht mit dem Pfad $[\{2\}, \{1, 3\}]$ empfängt.



Jede Nachricht ist ein Exemplar einer Klasse, die das Interface `Message` implementiert, und enthält zusätzlich zum Pfad evtl. noch weitere Attribute. Beachten Sie, dass der Identifier einer Verbindung lokal ist, d.h. dass verschiedene Knoten die gleichen VerbindungsIds haben können (die dann zu irgendwelchen anderen Knoten führen können). So haben sowohl $n0$ als auch $n3$ eine Verbindung mit VerbindungsId 2. Es können mehrere Verbindungen zum selben Knoten führen (siehe Verbindungen 1 und 2 in $n3$), aber für das Routing hier spielt das keine Rolle und erfordert keine Sonderbehandlung.

Für Nachrichten gibt es die Methode `Message.getPath()`, welche den Pfad (eine Liste von Integer Sets) zurückgibt. Die Elemente des ersten Sets geben an, über welche Verbindungen die Nachricht in einem Knoten weitergeleitet werden *könnte*. Es ist möglich, dass für eine Verbindung in dieser Menge (d.h., einer Zahl) kein Eintrag im `routingTable` vorhanden ist. Z.B. ist oben im Knoten $n3$ kein Eintrag mit VerbindungsId 3 im `routingTable`, obwohl die Zahl 3 im ersten Set in der Liste auftritt. Solche VerbindungsIds werden ignoriert. Sie dürfen aber annehmen, dass immer mindestens eine gültige Verbindung im Set ist. Die weiteren Elemente des Pfades bestimmen, wie die Nachricht vom nächsten (und folgenden) Knoten weitergeleitet wird. Daher wird vor dem Weiterleiten der Nachricht das erste Set des Pfades entfernt und eine neue Nachricht mittels `Message.withPath(List<Set<Integer> newPath)` erstellt.

Die Methode `RoutingNode.receive(Message msg)` ist bereits teilweise implementiert. In den folgenden Tasks implementieren Sie Hilfsmethoden, welche dafür verwendet wurden:

1. Implementieren Sie die Methode `RoutingNode.candidates(List<Set<Integer>> path)`, welche den Pfad einer Nachricht nimmt und die Menge der gültigen VerbindungsIds bestimmt. Eine Verbindung ist gültig wenn die VerbindungsId im ersten Set von `path` enthalten ist *und* es einen Eintrag in `routingTable` gibt. Im Knoten `n3` im obigen Beispiel gibt diese Methode das Set `{1}` zurück.
2. Implementieren Sie `RoutingNode.selectConnection(Set<Integer> candidates)`. Die Methode nimmt ein Set von gültigen VerbindungsIds und gibt den Identifier mit der höchsten Priorität zurück.
 - Für einen Routing Knoten *n* hat ein Identifier *a* eine höhere Priorität als ein Identifier *b*, wenn über die Verbindung von *a* weniger Nachrichten weitergeleitet wurden als über die Verbindung von *b*.
 - Wenn beide Verbindungen gleich oft verwendet wurden, dann hat der kleinere Identifier die höhere Priorität.

Zur Erinnerung: `candidates` ist nicht leer und enthält nur Identifier, die in der Routing Table von *n* vorkommen.

Implementieren Sie auch die Methode `RoutingNode.incrementCount(int id)`, welche für eine Verbindung (bestimmt durch den Parameter `id`) die Anzahl der weitergeleiteten Nachrichten um 1 erhöht. Sie dürfen annehmen, dass `id` in der Routing Table vorkommt. In der vorgegebenen Implementation von `receive` wird `incrementCount` aufgerufen bevor die Nachricht weitergeleitet wird (d.h. wenn die gewählte Verbindung den Identifier `id` und den Nachbarknoten *n* hat, dann wird zuerst `incrementCount(id)` und dann `n.receive(...)` aufgerufen).

3. Implementieren Sie die Methode `RoutingNode.process(Message msg)`, welche *zugestellte* Nachrichten verarbeitet. Wie in der Methode `RoutingNode.receive(Message msg)` implementiert, gilt eine empfangene Nachricht als zugestellt, wenn der Pfad der Nachricht leer ist. Wenn einem `RoutingNode` ein Exemplar der Klasse `UpdateMessage` zugestellt wird, dann wird in der Routing Table dieses Knotens ein Eintrag hinzugefügt. Die Attribute `UpdateMessage.newId` und `UpdateMessage.newNode` enthalten dafür die VerbindungsId und die Referenz des Nachbarknotens. Wenn der Identifier in der Routing Table bereits vorhanden ist, dann wird der Eintrag mit dem neuen Knoten überschrieben und die Anzahl der über diese Verbindung weitergeleiteten Nachrichten wird auf 0 gesetzt.

Bei allen anderen Arten von Nachrichten (die natürlich auch `Message` implementieren) passiert nichts.

4. Implementieren Sie zusätzlich die Klasse `CountingNode`. Die Exemplare dieser Klasse sollen genau wie `RoutingNode` Nachrichten weiterleiten und verarbeiten. Aber anders als `RoutingNode` ignoriert `CountingNode` zugestellte Exemplare der Klasse `UpdateMessage`. Wird einem `CountingNode` hingegen ein Exemplar der Klasse `IntMessage` zugestellt, dann soll der Wert des `CountingNode.sum` Attributes dieses Knotens um den Wert des Attributes `IntMessage.payload` erhöht werden. (Sie können davon ausgehen, dass `payload > 0` ist.)

Generische Listen (2020 W12)

In dieser Aufgabe implementieren Sie eine generische verkettete Liste. Anhang 52.1 zeigt eine generische Version eines Interfaces für Listen. Vervollständigen Sie die Klasse `MyListImpl`, sodass die Klasse das `MyList` Interface implementiert. Dem Interface wurden zwei neue Methoden hinzugefügt. Die Methode `MyListNode getNode(int index)` gibt den Knoten zurück, welcher den Wert der Liste an Position `index` speichert. `MyListNode` ist selber ein Interface (siehe Anhang 52.2) mit Methoden, welche jeweils den gespeicherten Wert des Knoten, den nächsten Knoten, und ob es einen nächsten Knoten gibt zurückgeben. Damit überprüfen wir, dass `MyListImpl` eine verkettete Liste implementiert. Die Methode `Iterator<T> iterator()` gibt einen Iterator für die Datenstruktur zurück. Implementieren Sie einen neuen Iterator, das heisst eine Klasse, welche das `Iterator` Interface implementiert, und geben Sie nicht den Iterator einer anderen Datenstruktur zurück (zum Beispiel den Iterator einer `ArrayList`). Dies können wir in den Tests der Korrektur testen. Die `void remove()` Methode vom Iterator wird nicht benötigt. Ein paar Tests finden Sie in `MyListTest`.

MyList Interface

```
public interface MyList<T> {

    /**
     * Return the value at position 'index'.
     * Throws a NoSuchElementException if the argument exceeds the list size.
     */
    public T get(int index);

    /**
     * Return the list node at position 'index'.
     * Throws a NoSuchElementException if the argument exceeds the list size.
     */
    public MyListNode<T> getNode(int index);

    /** Set the value at position 'index' to 'value'. */
    public void set(int index, T value);

    /** Returns whether the list is empty (has no values). */
    public boolean isEmpty();

    /** Returns the size of the list. */
    public int getSize();

    /** Inserts 'value' at position 0 in the list. */
    public void addFirst(T value);

    /** Appends 'value' at the end of the list. */
    public void addLast(T value);

    /** Appends the 'other' list to the end of the list. */
    public void addAll(MyList<T> other);
```

```

    /**
     * Removes and returns the first value of the list.
     * Throws a NoSuchElementException if the List is empty.
     */
    public T removeFirst();

    /**
     * Removes and returns the last value of the list.
     * Throws a NoSuchElementException if the List is empty.
     */
    public T removeLast();

    /** Removes all values from the list, making the list empty. */
    public void clear();

    /** Returns an iterator to the data structure. */
    public Iterator<T> iterator();
}

```

MyListNode Interface

```

public interface MyListNode<T> {

    /** Returns the value stored in the node. */
    public T value();

    /** Sets the value stored in the node. */
    public void setValue();

    /** Returns false iff this is the last node of the list. */
    public boolean hasNext();

    /** Returns next node. */
    public MyListNode<T> next();

    /** Sets the next node. */
    public void setNext();
}

```

Wörter (2019 W12)

In dieser Aufgabe implementieren Sie verschiedene Analysen von Wörtern eines Textes. Ein Wort eines Textes wird von der Klasse `MyWord` repräsentiert. Diese Klasse speichert sowohl das Wort als `String`, als auch alle Positionen des Wortes im Text. Zum Beispiel in dem Text "Seit Wochen wusste er, dass er die Prüfung nicht bestehen würde." hat das Wort "Seit" die einzelne Position 0 und "er" die Positionen 3 und 5. Zwei Wörter sind unterschiedlich, wenn die Wörter als unterschiedliche `Strings` im Text vorkommen. So sind "verlieren" und "verliert" unterschiedliche Wörter. Des

Weiteren definieren wir eine Ordnung auf Wörtern. Ein Wort x ist kleiner als ein Wort y , wenn der Abstand zwischen der ersten und letzten Position von x im Text kleiner ist als der Abstand zwischen der ersten und letzten Position von y im Text.

Ihre Aufgabe ist es nun, die Klasse `WordService` und ihre Analysen zu implementieren. Die Klasse hat einen Konstruktor, welche einen Text aus einem `WordScanner` ausliest und damit das Objekt initialisieren soll. Ein `WordScanner` funktioniert ähnlich wie `Scanner`, mit dem Unterschied, dass `next()` Satzzeichen aus dem nächsten String entfernt bevor es den String zurückgibt. Sie müssen die Wörter nicht weiter bearbeiten und dürfen annehmen, dass jeder von `next()` zurückgegebener String ein eigenes Wort ist.

- a) Implementieren Sie die Methode `getWords`, welche alle Wörter des Texts als `MyWord` Objekte zurückgibt.
- b) Implementieren Sie nun die Methode `top`, welche einen Integer `number` als Parameter nimmt und die `number` (Anzahl) grössten Wörter zurückgibt. Beachten Sie, dass Grösse durch unsere zuvor auf Wörtern definierte Ordnung bestimmt ist. Die Ergebnisliste muss absteigend sortiert sein, wobei die Reihenfolge von gleich grossen Worten egal ist.

Achtung: Ihre implementierten Methoden dürfen nur `IllegalArgumentException` werfen. Das Werfen anderer Exceptions ist verboten, unabhängig von den verwendeten Argumenten. Tests sind in der Datei `WordServiceTest.java` enthalten.

Palindrome (2018 W12)

In dieser Aufgabe schreiben Sie eine Methode `maxPalindrome`, die eine Liste von ganzen Zahlen entgegen nimmt, und das Set aller Palindrome mit maximaler Länge zurückgibt, welche Sublisten des Arguments sind.

- a) Erstellen Sie eine Klasse `Palindrome` mit einer Methode `maxPalindrome`. Diese Methode soll `public` und `static` sein und den Rückgabetyt `Set<List<Integer>>` haben. Die Methode hat einen Parameter. Der Typ dieses Parameters ist eine `List` von ganzen Zahlen. Sowohl `List` als auch `Set` sind aus dem `java.util`-Paket.

Wichtig: Erstellen Sie die `Palindrome`-Klasse in einem neuen Paket namens `"palindrome"` (welches kein Unterpaket eines anderen Pakets ist).

- b) Implementieren Sie nun die Methode, sodass sie das Set aller Palindrome mit maximaler Länge zurückgibt. Eine Liste aus Zahlen ist ein Palindrom, wenn die Liste und die Umkehrung der Liste äquivalent sind. Falls als Argument eine leere Liste, `null` oder eine Liste, die `null` enthält, übergeben wird, dann soll eine `RuntimeException` mit der Nachricht `"invalid list"` geworfen werden. Die übergebene Liste darf von der Methode natürlich nicht verändert werden.

Beispiel: Für die Liste `[13, 13, 4, 13, 4, 2]` ist das Set aller Palindrome, welche Sublisten sind: `{[13], [4], [2], [13, 13], [13, 4, 13], [4, 13, 4]}`. Das Set der Palindrome mit maximaler Länge ist dann: `{[13, 4, 13], [4, 13, 4]}`.