Praktikum Grundlagen der Programmierung

Prof. Dr. Harald Räcke, Prof. Dr. Felix Brandt, J. Kranz, A. Reuss WS 2017/18

Übungsblatt 6 Abgabefrist: 04.12.2017 um 05:00 Uhr

Σ Punkte: 20

Aufgabe 6.1 (P) Rekursion 1.0

Wir wollen uns in dieser Aufgabe mit dem Wesen(tlichen) der rekursiven Ausführung von Prozeduren befassen. Dazu wählen wir eine simple Aufgabe, die sich mittels einer `for`-Schleife leicht lösen lässt: die Ausgabe aller Zahlen von 1 bis n, wobei n vom Benutzer eingegeben wird. D. h. n steht zu Beginn noch nicht fest. Ein Programmgerüst ist mittels `Rekursion.java` gegeben.

In einer `for`-Schleife kann n z. B. einfach als Obergrenze gewählt werden, ähnlich in einer While-Schleife. Um die Aufgabe rekursiv zu lösen, benötigen wir eine Abbruchbedingung für die rekursiven Aufrufe.

- Versuchen Sie, das Verhalten der Prozedur `AusgabeMitSchleife` mittels einer rekursiven Umsetzung zu erreichen. Welche Probleme ergeben sich dabei? Wie könnte man diese lösen?
- Um zu veranschaulichen, welche Instanzen einer rekursiven Prozedur gerade aktiv sind, lassen Sie eine Ausgabe vor und nach dem rekursiven Aufruf machen. Lassen Sie dabei auch die *Rekursionstiefe* darstellen, indem die eigentliche Ausgabe um die entsprechende Anzahl Leerzeichen eingerückt wird. Die Ausgabe sollte so aussehen wie beim Aufruf der Prozedur `AusgabeTiefe`.

Aufgabe 6.2 (P) Wurzel berechnen

Das Heron-Verfahren (auch „babylonisches Wurzelziehen“ genannt) ist ein alter iterativer Algorithmus zur Bestimmung einer rationalen Näherung der Quadratwurzel $\sqrt{a}$ einer reellen Zahl a. Die Berechnungsvorschrift lautet:

$$x_{n+1} \quad = \quad \frac{1}{2} \cdot (x_n + \frac{a}{x_n})$$

Hierbei bezeichnet a die Zahl, deren Quadratwurzel bestimmt werden soll. Der Startwert x_0 ist eine beliebige positive Zahl.

Schreiben Sie ein *rekursives* Java-Programm `Wurzel.java`, das für eine einzulesende Zahl a die Quadratwurzel $\sqrt{a}$ näherungsweise nach dem Heron-Verfahren berechnet.

Das Verfahren soll abgebrochen und die Näherung ausgegeben werden, sobald das Quadrat der Näherung von a um weniger als einen Betrag ϵ abweicht, wobei ϵ vom Benutzer (zu Beginn) festgelegt wird.

Hinweis: Für die Implementierung könnte eine Hilfsfunktion hilfreich sein.

Aufgabe 6.3 (P) π

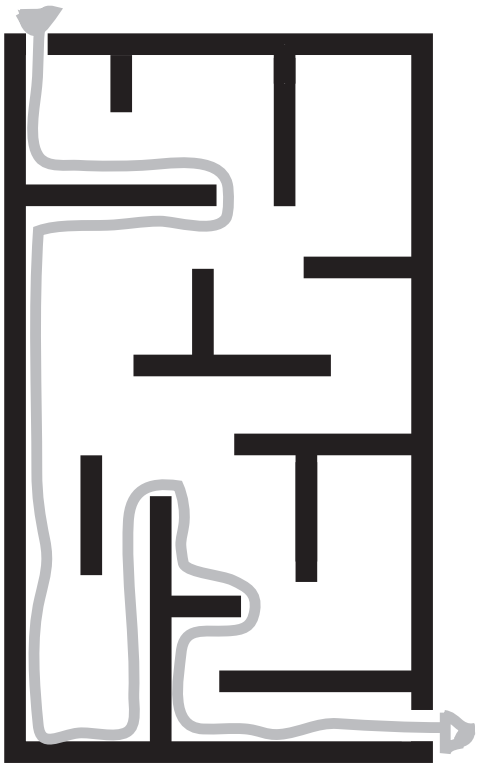
Gegeben sei folgende Rekursionsgleichung zur Annäherung von π:

$$f_n = \begin{cases} \frac{4}{2n+1} + f_{n-1} & n \text{ gerade} \\ \frac{-4}{2n+1} + f_{n-1} & n \text{ ungerade} \end{cases}$$

wobei $f_0 = 4$.

Schreiben Sie ein Programm, das nach Eingabe des Index n die Annäherung f_n für π *rekursiv* berechnet und ausgibt.

Aufgabe 6.4 (P) Der Weg zum Erfolg



Um in einem Labyrinth vom Eingang zum Ausgang zu kommen, kann man die sogenannte Rechte-Hand-Regel verwenden: Man tastet sich so an den Wänden entlang, dass man immer zu seiner rechten Seite eine Wand spürt. Dieser Algorithmus funktioniert allerdings nicht immer; falls man auf einer *Insel* im Labyrinth beginnt, läuft man im Kreis.

Startet man von einem Eingang, kommt man zumindest wieder dorthin zurück und kann ggf. feststellen, dass es keinen (weiteren) Ausgang gibt. Strenggenommen müsste der Eingang markiert werden, um ihn von einem (anderen) Ausgang zu unterscheiden, wenn man (wieder) hinkommt.

Im Folgenden soll das Durchwandern des Labyrinths mittels der Rechte-Hand-Regel mit Hilfe der Klasse `Maze` von der Praktikumswebseite implementiert werden. `Maze` bietet die Methode `int[][] generateMaze(int width, int height)`, die ein zufälliges Labyrinth mit `width` Spalten und `height` Zeilen erzeugt, repräsentiert als ein zweidimensionales `int`-Array. Hat dieses Array an der Stelle `[x][y]` den Wert

- `WALL`, befindet sich im Labyrinth bei den Koordinaten (x, y) eine Wand.
- `PLAYER`, befindet sich dort die Durchwanderin.
- `FREE` oder `OLD_PATH_DONE`, befindet sich dort nichts.

Alle diese Namen für unterschiedliche `int`-Konstanten sind in der Klasse `Maze` bereits definiert. Der Wert `OLD_PATH_DONE` zeigt an, dass der Durchwandernde die Stelle bereits besucht und wieder verlassen hat.

Der aktuelle Zustand des Labyrinths lässt sich mittels `draw(int[][] maze)` in einem Fenster grafisch darstellen. Der Eingang befindet sich links oben im Labyrinth bei den Koordination $(1, 0)$, d. h. einen Schritt rechts und null Schritte unterhalb vom oberen linken Eck $(0, 0)$. Der Ausgang ist rechts unten im Labyrinth bei den Koordinaten `(width-1, height-2)`.

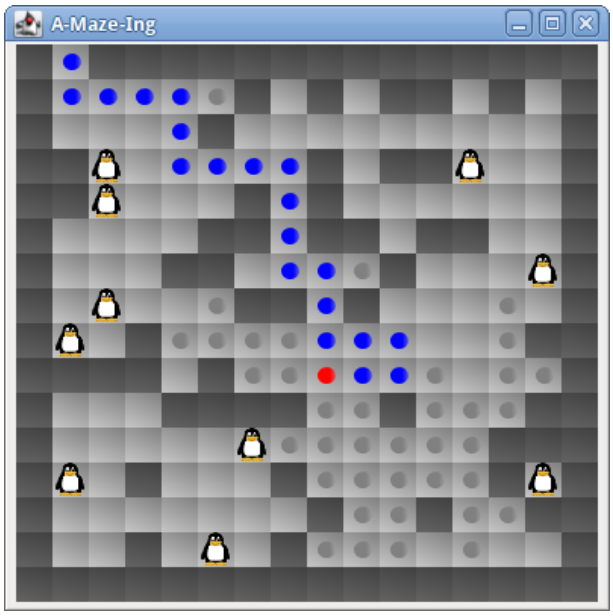
Ersetzen Sie `extends MiniJava` durch `extends Maze` und

- implementieren Sie eine Methode `void walk(int x, int y, int direction)`, welche vom Startpunkt `(1,0)` aus rekursiv alle Lösungsschritte der Rechte-Hand-Regel nacheinander ausführt und den neuen Zustand jeweils mittels `draw` ausgibt.
- implementieren Sie eine `main`-Methode, welche mittels `generateMaze` ein Labyrinth erzeugt und dann mittels `walk` nach dem Ausgang sucht. Es soll auch ausgegeben werden, ob ein Ausgang gefunden wurde oder nicht.

Hinweis: Zum Testen ist es nützlich, anstelle von `generateMaze` eine Funktion zu verwenden, die immer das selbe Labyrinth liefert. Eine solche Funktion ist mit `generateStandardMaze` in `Maze.java` enthalten. Testen Sie Ihr Programm auch damit!

Die Hausaufgabenabgabe erfolgt über Moodle. Bitte geben Sie Ihren Code als UTF8-kodierte (ohne BOM) Textdatei(en) mit der Dateiendung `.java` ab. Geben Sie **keine** Projektdateien Ihrer Entwicklungsumgebung ab. Geben Sie **keinen** kompilierten Code ab (`.class`-Dateien). Geben Sie Ihren Code **nicht** als Archiv (z.B. als `.zip`-Datei) ab. Nutzen Sie **keine** Ordner in Moodle. Nutzen Sie **keine** Packages. Achten Sie darauf, dass Ihr Code kompiliert. Bitte vermerken Sie aus Datenschützgründen nicht Ihren Namen oder Ihre Matrikelnummer im Code. Hausaufgaben, die sich nicht im vorgegebenen Format befinden, werden nur mit Punktabzug oder gar nicht bewertet.

Aufgabe 6.5 (H) Ohne Ausweg, aber mit Pinguin [6 Punkte]



In dieser Aufgabe geht es darum, möglichst vielen armen Pinguinen zu helfen. Die Pinguine sind zusammen mit Ihnen in einem Labyrinth eingeschlossen, aus dem es keinen Ausgang gibt[1]; Ihre Aufgabe ist es, die Pinguine zu suchen und zu füttern.

Dazu brechen Sie von Ihrem Startpunkt in alle möglichen Richtungen auf. Um möglichst viele Pinguine tragen zu können, bleibt das Futter allerdings am Ausgangspunkt, sodass Sie dorthin zurückfinden müssen. Um sich nicht zu verirren, wagen Sie sich niemals auf Felder, die auf keinem der 8 umgebenden Felder eine Wand aufweisen. Ihre Tierliebe ist darüber hinaus vom Abstand zum Ausgangspunkt beschränkt; den aktuell aktiven Weg halten Sie stets kleiner als eine gegebenen Maximaldistanz, um im Fall von plötzlich einsetzender Müdigkeit oder nachlassender Knuffigkeit der Pinguine jederzeit komfortabel zum Ausgangspunkt zurückkehren zu können.

Implementieren Sie die Java-Methode `public static int walk(int x, int y, int maxDistance)`, die ausgehend von einem Startpunkt rekursiv alle Wege nach traurigen Pinguinen absucht und die Anzahl der gefundenen Pinguine zurückliefert. Die Länge des *aktiven Weges*, also die Anzahl an Feldern, die schon besucht, aber noch nicht endgültig verlassen wurden, soll dabei stets kürzer `maxDistance` sein. Leiten Sie Ihre Klasse von `Maze` ab und nutzen Sie die Methode `void draw(int[][] labyrinth)`, um das Labyrinth nach jedem Schritt zu zeichnen. Das zweidimensionale Array enthält dabei für jedes Feld einen Wert, der speichert, was auf diesem Feld zu finden ist. Es kommen folgende Konstanten vor:

[1] Bitte bauen Sie diese Aufgabe aus Gründen des Tierschutzes nicht zuhause nach.

- `WALL`: An der Position im Labyrinth befindet sich eine Wand.
- `FREE`: An der Position im Labyrinth befindet sich nichts.
- `PLAYER`: An der Position im Labyrinth befindet sich die heroische Pinguinretterin.
- `OLD_PATH_ACTIVE`: Die Position wurde während Ihrer Pinguinsuche bereits besucht, Sie kehren zu dieser Position aber auf dem Rückweg zurück, um weitere Wege zu finden (die Position ist Teil des aktiven Weges).
- `OLD_PATH_DONE`: Die Position wurde während Ihrer Pinguinsuche bereits besucht, Sie kehren zu dieser Position nicht wieder zurück.
- `PENGUIN`: An der Position im Labyrinth befindet sich ein notleidender Pinguin.

Einige dieser Konstanten können Sie eine Position ins Feld schreiben (wie z.B. `PLAYER`), andere lesen Sie (wie z.B. `WALL`). Nehmen Sie gefundene Pinguine auf, indem Sie sie durch passende Pfad-Konstanten (`OLD_PATH_ACTIVE` bzw. `OLD_PATH_DONE`) ersetzen. Implementieren Sie außerdem ein Hauptprogramm, welches zunächst den Benutzer nach der Größe des Labyrinthes und seinen maximalen Abstand zum Startpunkt während der Suche fragt. Das Hauptprogramm nutzt dann die Methode `public static int[][] generatePenguinMaze(int width, int height)`, um ein Labyrinth zu generieren. Sie starten am Punkt $(1, 0)$. Führen Sie nun die Pinguinsuche durch und geben Sie schließlich aus, wie viele Pinguine Sie retten konnten.

Hinweise:

- Als Hilfestellung zur Nutzung des Labyrinthes ist die entsprechende P-Aufgabe sehr nützlich. Weitergehende Informationen zur Handhabung von Pinguinen erhalten Sie evtl. unter `http://www.hellabrunn.de/`.
- Sie können eine MiniJava-Methode `name()` durch den Aufruf `MiniJava.name()` nutzen.
- `Maze.generateStandardPenguinMaze` erzeugt für die übergebene Größe das immer gleiche Labyrinth.

Aufgabe 6.6 (H) Ponynome dritten Grades [6 Punkte]

In dieser Aufgabe sollen Sie ein Programm schreiben, das die Nullstellen eines Polynoms dritten Grades berechnen kann. Wir beschränken uns hierbei auf Polynome, deren Nullstellen ganzzahlig sind. Die Polynome haben also die folgende allgemeine Form:

$$f(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0 = a_3x^3 + a_2x^2 + a_1x + a_0$$

Die Benutzerin soll dazu die Parameter a_3, a_2, a_1 und a_0 eingeben. Ihr Programm berechnet zuerst eine Nullstelle mit Hilfe eines Näherungsverfahrens, reduziert das Polynom um einen Grad und berechnet für das reduzierte Polynom die verbleibenden Nullstellen per Lösungsformel (auch bekannt als Mitternachts- bzw. pq-Formel).

Im folgenden werden wir uns ein Beispiel ansehen, unser Polynom sei dazu $f(x) = 0.5x^3 + 5x^2 - 33.5x - 308$. Wir müssen nun sukzessive die Nullstellen dieses Polynoms bestimmen; das Verfahren dazu gliedert sich in einige Teilschritte.

Bestimmung eines passenden Intervalls

Wir beginnen mit Suche nach einem passenden Intervall für die Nullstellen.

1. Wir haben ein Intervall $[a, b]$ mit $a < 0 < b$.
2. Wir berechnen $f(a), f(b)$ und überprüfen, ob die beiden Werte unterschiedliche Vorzeichen haben.
3. Wenn dies **nicht** der Fall ist, so vergrößern wir die Intervallgrenzen um den Faktor 10, fahren also mit dem Intervall $[10 * a, 10 * b]$ fort.
4. Wenn sich die Vorzeichen unterscheiden, dann fahre mit dem nächsten Teilschritt fort.

Beispiel

1. Erstes Intervall $[-2, 2]$. $f(-2) = -225$ und $f(2) = -351 \rightarrow$ Neues Intervall: $[-20, 20]$.
2. Intervall $[-20, 20]$. $f(-20) = -1638$ und $f(20) = 5022$
   $\rightarrow$ Wir haben ein Intervall gefunden, in dem es mindestens eine Nullstelle gibt.

Bestimmung der ersten Nullstelle

Wir haben jetzt ein Intervall, in dem mindestens eine Nullstelle ist; nun versuchen wir, diese rekursiv zu bestimmen.

1. Wir haben ein Intervall $[a, b]$. Wir berechnen den Mittelpunkt m dieses Intervalls als $m = \frac{a+b}{2}$.
2. Wir berechnen $f(a), f(b), f(m)$. Falls $f(a) = 0$, $f(b) = 0$ oder $f(m) = 0$, so haben wir eine Nullstelle gefunden.
3. Wir prüfen, ob die Werte $f(a)$, $f(b)$ bzw. $f(m)$ positiv oder negativ sind. Wenn $f(a)$ und $f(m)$ unterschiedliche Vorzeichen haben, so gibt es in diesem Bereich eine Nullstelle. Dasselbe gilt für $f(m)$ und $f(b)$.
4. Wir passen das Intervall an, sodass die Nullstelle weiterhin im Intervall liegt. Ist die Nullstelle zwischen a und m, dann ist unser neues Intervall $[a, m]$, ansonsten $[m, b]$.

Beispiel

1. Intervall $[-20, 20]$. $m = \frac{-20+20}{2} = 0$.
   $f(-20) = -1638$, $f(0) = -308$ und $f(20) = 5022$
   $\rightarrow$ Neues Intervall $[0, 20]$
2. Intervall $[0, 20]$. $m = \frac{0+20}{2} = 10$.
   $f(0) = -308$, $f(10) = 357$ und $f(20) = 5022$
   $\rightarrow$ Neues Intervall $[0, 10]$

3. Intervall $[0, 10]$. $m = \frac{0+10}{2} = 5$.
   $f(0) = -308$, $f(5) = -288$ und $f(10) = 357$
   $\rightarrow$ Neues Intervall $[5, 10]$
4. Intervall $[5, 10]$. $m = \frac{5+10}{2} = 7.5 \rightarrow 7$.
   $f(5) = -288$, $f(7) = -126$ und $f(10) = 357$
   $\rightarrow$ Neues Intervall $[7, 10]$
5. Intervall $[7, 10]$. $m = \frac{7+10}{2} = 8.5 \rightarrow 8$.
   $f(7) = -126$, $f(8) = 0$ und $f(10) = 357$
   $\rightarrow$ Nullstelle bei $x = 8$.

Reduktion des Polynoms

Wir haben nun eine Nullstelle gefunden und müssen das Polynom mit dieser Nullstelle reduzieren. Dafür benutzen wir das Horner-Schema[2]. Das Horner-Schema ist hier eine Tabelle mit 3 Zeilen und 5 Spalten.

In der obersten Zeile stehen – beginnend mit der 2. Spalte – die Exponenten in absteigender Reihenfolge. In der 2. Zeile stehen die jeweiligen Koeffizienten. In der 3. Zeile steht am Anfang nur ein Eintrag in der ersten Spalte und zwar $x = x_0$, wobei x_0 der Wert der bereits berechneten Nullstelle ist.

Tabelle 1: Horner-Schema: Anfangsformat

	x^3	x^2	x^1	x^0
	0.5	5	-33.5	-308
x=8				

Nun muss man das Horner-Schema anwenden.

1. Zuerst wird der Koeffizient von x^3 in die 2. Zeile der x^2-Spalte geschrieben.
2. Um die verbleibenden Werte zu berechnen, addieren wir den Wert aus der 2. Zeile der vorherigen Spalte mit dem Produkt des x-Wertes mit dem Wert der 3. Zeile der vorherigen Spalte.
3. Wenn es eine echte Nullstelle war, so gibt es am Ende keinen Rest.

Tabelle 2: Horner-Schema: Fertige Tabelle

	x^3	x^2	x^1	x^0
	0.5	5	-33.5	-308
x=8		0.5	9	38.5

Das reduzierte Polynom kann aus der Tabelle abgelesen werden und lautet $f(x) = 0.5x^2 + 9x + 38.5$.

Bestimmung der übrigen Nullstellen mittels Lösungsformel

Für dieses Polynom können wir die bereits aus der Schule bekannte Lösungsformel benutzen.

[2] `https://de.wikipedia.org/wiki/Horner-Schema`

$$x_{1,2} = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}{2a_2}$$

Für unser Beispiel folgt also:

$$x_{1,2} = \frac{-9 \pm \sqrt{9^2 - (4 * 0.5 * 38.5)}}{2 * 0.5} = \{-11, -7\}$$

Gehen Sie bei Ihrer Implementierung wie folgt vor:

1. Implementieren Sie die Java-Methode `public static int[] quadraticFormula (double[] coefficients)`. Die Methode erwartet ein Array der Länge 3 mit den Koeffizient in der Reihenfolge a_2, a_1, a_0 und gibt ein Array mit allen möglichen Lösungen zurück. **Hinweis**: Wenn es nur eine oder gar keine Lösung gibt, soll das Array die entsprechende Länge haben.

2. Implementieren Sie eine Java-Methode `public static double[] hornerSchema (double[] coefficients, int x0)`. Die Methode erwartet ein Array der Länge 4 in der Reihenfolge a_3, a_2, a_1, a_0 und gibt ein Array der Länge 3 zurück, welches die Koeffizienten des reduzierten Polynoms enthält.

3. Implementieren Sie die Methode `public static double calculateY(double[] coefficients, int x)`, die für ein Polynom des Grades $n \leq 3$ den Y-Wert berechnet. Die Methode erwartet ein Array der **maximalen** Länge 4. Die Koeffizienten sind in absteigender Reihenfolge. Die Methode gibt den Wert der Funktion an Stelle x zurück.

4. Implementieren Sie anschließend eine **rekursive** Java-Methode `public static int[] findIntervalRecursive(double[] coefficients, int a, int b, int factor)`. Die Methode vergrössert die Intervallgrenzen rekursiv solange um einen festen Faktor $factor$, bis die Funktion an den Grenzen unterschiedliche Vorzeichen aufweist.

5. Implementieren Sie anschließend eine **rekursive** Java-Methode `public static int findRootRecursive(double[] coefficients, int a, int b)`. Die Methode sucht nach einer Nullstelle der übergebenen Funktion im Intervall $[a, b]$.

6. Schreiben Sie zm Schluss ein Rahmenprogramm, dass vom Benutzer die Eingabe von 4 `double`-Variablen erwartet. Danach soll Ihr Programm für dieses Polynom die Nullstellen berechnen, ihre Anzahl und ihren Wert ausgeben.

Hinweis: Fehlerhafte Argumente müssen bei den Methoden nicht abgefangen werden.

Hinweis: Sie dürfen und sollen Methoden der Math-Bibliothek von Java verwenden.

Aufgabe 6.7 (H) Von der Blase zur Latex-Base [8 Punkte]

Auf dem letzten Blatt wurde das Rechnen zu verschiedenen Zahlenbasen geübt. Auf diesem Blatt werden wir dieses Wissen nutzen, um die Lösung solcher Rechnungen zu automatisieren. Dazu soll die schriftliche Addition und Multiplikation zu eine beliebigen Basis implementiert werden.

Wir repräsentieren eine Zahl zur Basis b als Array von Ziffern, wobei die niederwertigste Ziffer an Index 0 im Array zu finden ist. Jede Ziffer wiederum ist eine Zahl zwischen 0 und $b-1$. Auf diese Weise steht z.B. das Array `{1, 12, 15, 3}` für die Zahl $3FC1_{16}$.

Das Programm soll zu Beginn die Benutzerin um Eingabe zweier Zahlen und einer Basis bitten. Es soll anschließend die schriftliche Multiplikation als Latex-Tabelle ausgeben. Gibt der Benutzer z.B. die Zahlen `12398` und `189` zur Basis 16 ein, so soll folgende Ausgabe erscheinen:

```
\begin{tabular}{cccccccccc}
&1 & 2 & 3 & 9 & 8 &  $\ast$  & 1 & 8 & 9\\
\hline
+&&&&& A& 4& 0& 5& 8\\
+&&&& 9& 1& C& C& 0& 0\\
+&&& 1& 2& 3& 9& 8& 0& 0\\
\hline
=&&& 1& B& F& A& 4& 5& 8\\
\end{tabular}
```

In ein Latex-Dokument eingebunden kann die Rechnung gut nachvollzogen werden:

	1	2	3	9	8	$*$	1	8	9
+					A	4	0	5	8
+				9	1	C	C	0	0
+			1	2	3	9	8	0	0
=			1	B	F	A	4	5	8

Ein Beispiel für ein komplettes Latex-Dokument, welches obige Tabelle enthält, finden Sie unter `https://sharelatex.tum.de/project/5a0ddc835e7fdc7300ce8435`. Bearbeiten Sie die folgenden Teilaufgaben.

1. Implementieren Sie die Methode `public static int[] readNumber()`, die eine Zahl von der Benutzerin einliest und als Array von Ziffern entsprechend obiger Beschreibung zurückliefert. Die Benutzerin soll eine Zahl zu einer beliebigen Basis zwischen 2 und 36 eingeben können. Ziffern über 9 werden durch Großbuchstaben zwischen A und Z repräsentiert. Gibt der Benutzer eine ungültige Zahl ein, so soll er um erneute Eingabe gebeten werden.

2. Implementieren Sie eine Java-Methode `public static String toString(int[] number)`, die eine Zahl im oben beschriebenen Format in einen String umwandelt, welcher die Zahl in einem für Menschen gut lesbaren Format enthält. Nutzen Sie diese Methode für Debug-Ausgabe während der Entwicklung.

3. Implementieren Sie die Methode `public static int[] add(int[] a, int[] b, int base)`, die zwei Zahlen zu einer gegebenen Basis addiert und das Ergebnis zurückliefert. Die Argumente sollen nicht verändert werden.

4. Implementieren Sie die Methode `public static int[] mulDigit(int[] a, int digit, int shift, int base)`, die eine Zahl mit einer Ziffer multipliziert und um `shift` viele Stellen nach links verlagert. Es soll also $\texttt{a} * \texttt{base}^{\texttt{shift}} * \texttt{digit}$ berechnet werden. Das Argument-Array soll nicht verändert werden.

5. Implementieren Sie die Methode `public static int[] mul(int[] a, int[] b, int base)`, die zwei Zahlen zu einer gegebenen Basis multipliziert und das Ergebnis zurückliefert. Die Argumente sollen nicht verändert werden.

6. Implementieren Sie ein Hauptprogramm.

7. Erweitern Sie Ihr Programm um die Ausgabe der Latex-Tabelle. Sie können dazu selbst Methoden hinzufügen oder bestehende Methoden um eine Ausgabe ergänzen.

Hinweis: Rechnen Sie stets in der jeweiligen Basis, wandeln Sie die Zahl nicht erst ins Dezimalsystem um. Wandeln Sie außerdem die Zahl nicht zwischenzeitlich in einen Integer um.

Hinweis: Es kann hilfreich sein, bei der Implementierung in folgenden Schritten vorzugehen; Lösungen, die nur einige der Teilschritte umsetzen, bringen selbstverständlich auch Punkte:

1. Implementieren Sie zunächst Ihr Programm mit einer textuellen Ausgabe der Lösung, welches nur die Multiplikation zur Basis 10 beherrscht.

2. Erweitern Sie Ihr Programm für die Nutzung beliebiger Basen.

3. Ersetzen Sie schließlich die textuelle Ausgabe der Lösung durch eine schrittweise Ausgabe des Latex-Codes.