Praktikumsaufgabe 3 Teil-1

3.1 Eindimensionale Arrays

1. Schreiben Sie die Methode `alle_vielfache_von?(ary,n)`, die für ein 1-dimensionales Array `ary` prüft, ob alle Elemente des Arrays Vielfache von `n` sind. Die Methode gibt einen booleschen Wert zurück. Die Verwendung der Methode `all?` von `Array` ist nicht erlaubt. Wenn `ary` leer ist, dann soll das Ergebnis `true` sein.
2. Schreiben Sie die Methode `sammle_strings_der_laenge(ary,n)`, die aus einem 1-dimensionalen Array `ary` von Zeichenketten nur die Zeichenketten einsammelt, die die Länge `n` haben. Die Verwendung der Methoden `select`, `find_all` u.ä. ist nicht erlaubt.
3. Schreiben Sie die Methode `casecmp(ary1,ary2)`, die für zwei Arrays von Zeichenketten berechnet, ob `ary1` kleiner, gleich oder größer als `ary2` ist. Die Methode gibt -1 (für `ary1<ary2`), 0 (für `ary1==ary2`), +1 (für `ary1 > ary2`) zurück. Beim Vergleich der Zeichenketten soll Groß- und Kleinschreibung vernachlässigt werden. Das erste Element, in dem sich `ary1` und `ary2` unterscheiden, entscheidet das Ergebnis. Lösen Sie die Aufgabe mit indiziertem Zugriff auf die Elemente der Arrays und einer geeigneten Methode für den Zeichenketten-Vergleich. **Achtung**: Die Arrays müssen nicht gleich lang sein.
4. Schreiben Sie die Methode `sechs_aus_49()`, die das Ziehen der 6 Lottozahlen simuliert. Die Methode geht wie folgt vor.
    a. Zu Beginn wird ein Array `zahlen` mit den 49 Zahlen erzeugt.
    b. Dann würfelt die Methode einen Index aus, liest über den Index das Element aus dem Array `zahlen`, merkt sich das Element in einem Ergebnis-Array und löscht das Element in `zahlen`.
    c. Schritt b. wird 6-mal wiederholt.
    d. **Achtung**: Achten Sie beim Würfeln darauf, dass Sie immer einen gültigen Index auswürfeln.
5. Schreiben Sie die Methode `shuffle(ary)`, die die Elemente in `ary` in eine beliebige Reihenfolge bringt und als Ergebnis das Array mit den beliebig angeordneten Elementen zurückgibt. Verwenden Sie für die Lösung einen Zufallszahlen-Generator. Das Benutzen der Methode `shuffle` und ähnlicher Methoden der Klasse Array ist **nicht erlaubt**.

Wenn Sie alle Methoden implementiert haben, dann sollte das Ausführen des nachfolgenden Scripts die gezeigten Ergebnisse liefern. Die Ergebnisse für 4 und 5 dürfen in der Reihenfolge abweichen. Das Ergebnis für 4 auch im Inhalt, aber nicht der Anzahl der Elemente.

```
require "./arrays"
ary1 = [1,2,3,4,5,6,7]
ary3 = []
(0..20).step(3){|elem| ary3 << elem}

ary_strings_1 = %w{das ist ein Array mit vielen Strings der Laenge drei}
ary_strings_2 = %w{Das ist ein ARRAY mit vielen STRINGS der Laenge drei}
ary_strings_3 = %w{Das ist ein ARRAY mit vielen STRINGS}
ary_strings_4 = %w{Xenon}

puts "alle_vielfache_von?"
puts alle_vielfache_von?(ary1,2)
puts alle_vielfache_von?(ary3,3)

puts "sammle_strings_der_laenge"
p sammle strings der laenge(ary strings 1,3)
p sammle_strings_der_laenge(ary_strings_1,6)
p sammle_strings_der_laenge(ary_strings_1,10)
p sammle_strings_der_laenge([], 20)

puts "casecmp"
puts casecmp(ary_strings_1,ary_strings_2)
puts casecmp(ary_strings_1,ary_strings_3)
puts casecmp(ary_strings_3,ary_strings_1)
puts casecmp(ary_strings_1,ary_strings_4)

puts "sechs_aus_49"
srand(9999)
p sechs_aus_49()
p sechs_aus_49()

puts "shuffle"
p shuffle(ary1)
p shuffle(ary_strings_1)
p shuffle([])
```

Ergebnisse:

```
alle_vielfache_von?
false
true
sammle_strings_der_laenge
["das", "ist", "ein", "mit", "der"]
["vielen", "Laenge"]
[]
[]
casecmp
0
1
-1
-1
sechs_aus_49
[23, 49, 47, 30, 40, 27]
[39, 25, 17, 13, 45, 37]
shuffle
[7, 6, 4, 2, 1, 3, 5]
["Strings", "mit", "Array", "der", "Laenge", "das", "ist", "ein",
"drei", "vielen"]
[]
```

3.2 Testen ausgewählter Methoden aus 3.0

Schreiben Sie für die Methoden 3.0.1-3.0.3 Unit-Tests. Schreiben Sie je eine Testmethode pro Methode und die positiven, negativen und Randfälle.

Also z.B. für die Methode `casecmp`:

- `test_casecmp_pos`
- `test_casecmp_neg`
- `test_casecmp_grenze`

3.3 Rechnen mit Matrizen

Sie sollen die Klasse `Matrix` um die folgenden Methoden erweitern:

1. `initialize(zeilen, spalten)`: Initialisiert die Matrix als 2-dimensionales Array mit Anzahl `zeilen` und Anzahl `spalten` und belegt alle Elemente mit dem Wert 0.
2. `kopiere_aus_array(ary)`: Kopiert aus einem unregelmäßigen 2-dimensionalen Array die Inhalte in die Matrix. Achtung: `ary` kann eine kleinere /größere Anzahl an Zeilen und Spalten als die Matrix haben.
3. `kopiere_in_array()`: Kopiert den Inhalt der Matrix in eine 2-dimensionales Array und gibt dieses als Ergebnis zurück. Es müssen alle Zeilen **und** Spalten kopiert werden.
4. `kopiere()`: Erzeugt eine Kopie der Matrix. Nutzen Sie dazu die Methoden aus 2. und 3.
5. `zwei_dim?(ary)`: Interne Methode, die für `kopiere_aus_ary` gebraucht wird. Die Methode prüft, ob `ary` ein 2-dimensionales Array ist, also ob in jeder Zeile wieder ein Array steht. Die enthaltenen Arrays müssen nicht die gleiche Länge haben.
6. `zeilen()`: Berechnet die Anzahl der Zeilen der Matrix
7. `spalten()`: Berechnet die Anzahl der Spalten der Matrix
8. `setze_element(zeile, spalte, wert)`: setzt `wert` an Position [`zeile`][`spalte`] des internen Arrays der Matrix.
9. `lese_element(zeile, spalte)`: Liest den Wert an Position [`zeile`][`spalte`] des internen Arrays der Matrix.
10. `+(other_matrix)`: Addiert die Matrix mit `other_matrix`. Die Operation darf `self` nicht verändern. Es muss also auf einer Kopie von `self` gearbeitet werden.
11. `*(skalar)`: Multipliziert die Matrix mit einem Skalar (einer Zahl). Die Operation darf `self` nicht verändern. Es muss also auf einer Kopie von `self` gearbeitet werden.
12. `-(other_matrix)`: Subtrahiert `other_matrix` von der Matrix. Lässt sich auf die Methoden aus 10. und 11. zurückführen. Die Operation darf `self` nicht verändern. Es muss also auf einer Kopie von `self` gearbeitet werden.
13. `laengstes_element()`: Berechnet das Element mit der größten Anzahl an Ziffern. Wird in `to_s()` benötigt.
14. `to_s()`: Bereitet den Inhalt der Matrix als Zeichenkette auf. In der Zeichenkette stehen die Zeilen untereinander. Alle Elemente belegen gleich viel Platz, werden rechtsbündig ausgerichtet (Methode `rjust` von `String`) und sind durch mindestens 1 Leerzeichen voneinander getrennt. Der benötigte Platz hängt von der Anzahl der Zeichen des längsten Elements ab (vgl. 13.).

Das Script `matrix_szenarien` enthält eine Reihe von Beispielaufrufen. Hier nun das Script und die Ausgaben des Scripts, wenn alles implementiert ist:

```
require './matrix'

srand(17999)
m1 = Matrix.new(3,4)
m2 = Matrix.new( 3,4)
ary = Array.new(5){|row| Array.new(rand(1..7)){|col|
(row+col)*rand(2..7)}}.shuffle
puts "m1=Matrix.new(3,4)"
puts "m1\n#{m1}"
puts "m1.kopiere_aus_array(#{ary})"
m1.kopiere_aus_array(ary)
puts "m1:\n#{m1}"
puts "m1 zeilen:#{m1.zeilen} spalten:#{m1.spalten}"
puts "m1 #{m1.kopiere_in_array()}"
puts "zwei_dim?(#{ary}):#{m1.zwei_dim?(ary)}"
m2.kopiere_aus_array(ary.shuffle.shuffle)
puts "m2\n#{m2}"
puts "m1+m2:\n#{m1.+(m2)}"
puts "m1-m2:\n#{m1-m2}"
puts "m1*2\n#{m1*2}"
puts "m1.laengstes_element():#{m1.laengstes_element}"
puts "m1.setze_element(1,1,99999)"
m1.setze_element(1,1,99999)
puts "m1\n#{m1}"
puts "m1.laengstes_element():#{m1.laengstes_element}"
puts "m1.lese_element(1,1):#{m1.lese_element(1,1)}"
```

Ausgabe:

```
m1=Matrix.new(3,4)
m1
0 0 0 0
0 0 0 0
0 0 0 0
m1.kopiere_aus_array([[18, 28, 20, 18, 21, 32], [3, 8, 15, 12, 25], [0, 5, 4, 21, 24], [10, 6], [12, 15, 42]])
m1:
18 28 20 18
 3  8 15 12
 0  5  4 21
m1 zeilen:3 spalten:4
m1 [[18, 28, 20, 18], [3, 8, 15, 12], [0, 5, 4, 21]]
zwei_dim?([[18, 28, 20, 18, 21, 32], [3, 8, 15, 12, 25], [0, 5, 4, 21, 24], [10, 6], [12, 15, 42]]):true
m2
 0  5  4 21
10  6  0  0
12 15 42  0
m1+m2:
18 33 24 39
13 14 15 12
12 20 46 21
m1-m2:
 18  23  16  -3
 -7   2  15  12
-12 -10 -38  21
m1*2
36 56 40 36
 6 16 30 24
 0 10  8 42
m1.laengstes_element():18
m1.setze_element(1,1,99999)
m1
   18    28    20    18
    3 99999    15    12
    0     5     4    21
m1.laengstes_element():99999
m1.lese_element(1,1):99999
```

Weiterer Hinweis: Die Klasse Matrix enthält eine Reihe von check_ Methoden, die z.B. Wertebereiche prüfen. Sie können die Methoden in Ihren Methoden verwenden!

3.4 Optionaler Teil

Sie sollen die Klasse Matrix um die Methode `mat_mult(other)` erweitern.

Die Methode multipliziert zwei Matrizen und gibt die Ergebnismatrix zurück. Die Ergebnismatrix hat so viel Zeilen wie die erste Matrix (also `self`) und so viel Spalten wie `other`.

Multiplikation von 2 Matrizen ist nur dann möglich, wenn die Zeilenlänge der 1'ten Matrix gleich der Spaltenlänge der 2'ten Matrix ist. In der vorbereiteten Methode wird dies bereits geprüft.

Eine einfache Art zwei Matrizen miteinander zu multiplizieren ist die Multiplikation (genauer das Skalarprodukt) der Zeilenvektoren der ersten Matrix mit den Spaltenvektoren der zweiten Matrix.

Dazu sollen zunächst in der Klasse `Vektor` die folgenden Methoden implementiert werden (*das sind natürlich nicht alle Methoden eines Vektors, sondern nur die, die für die Aufgabe benötigt werden.*):

1. `initialize(ary)`: Erzeugt einen Vektor aus den Werten des Arrays. Der Vektor wird intern als Array verwaltet. **Kopieren** Sie den Inhalt in das interne Array.
2. `size()`: Die Anzahl der Elemente des Vektors.
3. `[](index)`: Elementzugriff, gibt das Element an Position `index` zurück.
4. `skalar_produkt(vektor)`: Berechnet das Skalarprodukt zweier Vektoren.

Wenn die Klasse `Vektor` mit den minimal benötigten Methoden implementiert ist, können wir diese in der Methode `mat_mult` nutzen.

`mat_mult` muss jetzt jeden Zeilenvektor aus `self` mit jedem Spaltenvektor aus `other` „multiplizieren" und das Ergebnis des Skalarproduktes in die Zielmatrix schreiben. Das Ergebnis des Skalarproduktes des Zeilenvektors der Zeile `zeile` in `self` mit dem Spaltenvektor der Spalte `spalte` in `other` steht in der Ergebnismatrix auf Position [`zeile`][`spalte`].

Final benötigen wir also für die Matrix noch die Methoden:

1. `zeilen_vektor(zeile)`: Liefert einen Vektor mit Inhalt der `zeile` der Matrix.
2. `spalten_vektor(spalte)`: Liefert einen Vektor mit dem Inhalt der `spalte` der Matrix.

Hinweis: Informieren Sie sich in den einschlägigen Medien über die Definitionen für Matrixmultiplikation und Skalarprodukt von Vektoren, wenn Ihnen die Definitionen nicht bekannt sind.
Hinweis: Der optionale Aufgabenteil ist nicht Gegenstand der Abnahme, kann aber in meinen Sprechstunden besprochen werden.

Das Script `matrix_optional_szenarien` und die Ausgabe:

```ruby
require "./vektor"
require './matrix'

srand(17999)
m1 = Matrix.new(3,4)
ary = Array.new(5){|row| Array.new(rand(1..7)){|col| (row+col)*rand(2..7)}}.shuffle
puts "m1=Matrix.new(3,4)"
puts "m1\n#{m1}"
puts "m1.kopiere_aus_array(#{ary})"
m1.kopiere_aus_array(ary)
puts "m1:\n#{m1}"
puts "m3=Matrix.new(4,5)"
m3 = Matrix.new( 4,5)
ary3 = Array.new(4){Array.new(5){rand(3)}}
m3.kopiere_aus_array(ary3)
puts "m3\n#{m3}"
puts "m3.kopiere_aus_array(#{ary3})"
puts "m3\n#{m3}"
puts "vektor1_2=m1.zeilen_vektor(2):#{vektor1_2=m1.zeilen_vektor(2)}"
puts "vektor3_3=m3.spalten_vektor(3):#{vektor3_3=m3.spalten_vektor(3)}"
puts "vektor1_2.skalar_produkt(vektor3_3):#{vektor1_2.skalar_produkt(vektor3_3)}"
puts "m1.mat_mult(m3):\n#{m1.mat_mult(m3)}"
```

```
m1=Matrix.new(3,4)
m1
0 0 0 0
0 0 0 0
0 0 0 0
m1.kopiere_aus_array([[18, 28, 20, 18, 21, 32], [3, 8, 15, 12, 25], [0, 5, 4, 21, 24], [10, 6], [12, 15, 42]])
m1:
18 28 20 18
 3  8 15 12
 0  5  4 21
m3=Matrix.new(4,5)
m3
0 0 2 0 1
1 0 2 0 0
1 0 1 2 1
1 0 1 2 1
m3.kopiere_aus_array([[0, 0, 2, 0, 1], [1, 0, 2, 0, 0], [1, 0, 1, 2, 1], [1, 0, 1, 2, 1]])
m3
0 0 2 0 1
1 0 2 0 0
1 0 1 2 1
1 0 1 2 1
vektor1_2=m1.zeilen_vektor(2):(0,5,4,21)
vektor3_3=m3.spalten_vektor(3):(0,0,2,2)
vektor1_2.skalar_produkt(vektor3_3):50
m1.mat_mult(m3):
 66   0 130  76  56
 35   0  49  54  30
 30   0  35  50  25
```