1 2048

1.1 Aufgabe

Implementiere eine Funktion `repeat`. Dieser wird ein String und eine ganze Zahl n übergeben. Sie gibt einen String zurück, der aus n Wiederholungen des übergeben Strings besteht.

```
{.kotlin .cb-nb number_lines = false} repeat("Hello",3)
```

```
1 repeat("a",5)
```

```
aaaaa
```

1.2 Aufgabe

Implementiere eine Funktion `showNumber`. Dieser wird eine positive ganze Zahl übergeben. Du kannst davon ausgehen, dass diese maximal 6 Stellen hat. Die Funktion gibt einen String zurück, in dem die Zahl rechtsbündig dargestellt wird. Links steht immer ein `|` und mindestens ein Leerzeichen. Der String der zurückgegeben wird, besteht immer aus 8 Zeichen.

```
1 showNumber(1)
```

```
|      1
```

```
1 showNumber(2048)
```

```
|   2048
```

```
1 showNumber(999999)
```

```
| 999999
```

Hinweis: Nutze `repeat`!

1.3 Aufgabe

Implementiere eine Funktion `showRow`. Dieser wird eine Liste von Integern übergeben. Sie gibt einen String zurück, in dem alle Zahlen in der Liste, wie in der letzten Aufgabe dargestellt werden. Auch ganz rechts steht ein `|`. Zwischen einer Zahl und einem `|` ist immer mindestens ein Leerzeichen Abstand.

```
1 showRow(listOf(131072, 4, 2))
```

```
| 131072 |      4 |      2 |
```

```
showRow(listOf(8, 32, 128, 2))
```

```
|      8 |     32 |    128 |      2 |
```

Hinweis: Nutze `showNumber`!

1.4 Aufgabe

Implementiere eine Funktion `showBoard`. Dieser wird eine 2D-Liste mit Integern übergeben. Sie stellt jede Zeile wie in der letzten Aufgabe dar und gibt die Ergebnisse als Liste von Strings zurück.

```
showBoard(listOf(listOf(32, 2),listOf(4, 128)))
```

```
[|     32 |      2 |, |      4 |    128 |]
```

```
showBoard(
                listOf(
                    listOf(64, 2048, 2),
                    listOf(4, 2, 2048),
                    listOf(64, 2048, 2)
                )
            )
```

```
[|    64 |   2048 |     2 |, |     4 |     2 |   2048 |, |    64 |   2048 |     2 |]
```

Hinweis: Nutze `showRow`!

1.5 Aufgabe

Implementiere eine Funktion `printBoard`. Dieser wird eine 2D-Liste mit Integern übergeben. Sie stellt diese wie in der letzten Aufgabe dar und gibt sie an der Konsole aus.

```
printBoard(listOf(listOf(32, 2),listOf(4, 128)))
```

```
|     32 |      2 |
|      4 |    128 |
```

```
printBoard(listOf(
                    listOf(64, 2048, 2),
                    listOf(4, 2, 2048),
                    listOf(64, 2048, 2)
                ))
```

```
|     64 |   2048 |      2 |
|      4 |      2 |   2048 |
|     64 |   2048 |      2 |
```

Hinweis: Nutze `printBoard`!

1.6 Aufgabe

Implementiere eine Funktion `removeZeros`. Dieser wird eine Liste mit Integern übergeben. Sie gibt eine Kopie der Liste ohne 0-en zurück.

```
1 removeZeros(listOf(0, 1, 5, 0, 2, 4, 0))
```

```
[1, 5, 2, 4]
```

```
1 removeZeros(listOf(0, 1, 0, 2, 0))
```

```
[1, 2]
```

1.7 Aufgabe

Implementiere eine Funktion `fill`. Dieser wird eine Liste mit Integern, ein einzelnes Integer und eine gewünschte Länge übergeben. Du kannst davon ausgehen, dass diese Länge mindestens so hoch ist, wie die Länge der Liste.

Die Funktion gibt eine Liste mit der gewünschten Länge zurück. Dafür wird die übergebene Liste rechts mit dem übergebenen Integer ergänzt.

```
1 fill(listOf(0, 1), 2, 3)
```

```
[0, 1, 2]
```

```
1 fill(listOf(3), 10, 5)
```

```
[3, 10, 10, 10, 10]
```

```
1 fill(listOf(3, 4), 0, 2)
```

```
[3, 4]
```

1.8 Aufgabe

Implementiere eine Funktion `moveRowLeftHelper`. Dieser wird eine Liste mit Integern übergeben. Du kannst davon ausgehen, dass diese nur 2-er Potenzen, die mindestens 2 sind, enthält.

Die Funktion gibt die Liste der 2-er Potenzen zurück, die beim Kippen nach Links im Spiel 2048 entsteht.

```
1 moveRowLeftHelper(listOf(2))
```

```
[2]
```

Dabei werden zwei gleiche Zahlen zu einer doppelt so großen Zahl kombiniert.

```
1 moveRowLeftHelper(listOf(2, 2))
```

```
[4]
```

Wenn eine Zahl beim Kombinieren von zwei Zahlen entsteht, kann diese nicht nochmal kombiniert werden.

```
1 moveRowLeftHelper(listOf(2, 2, 4))
```

```
[4, 4]
```

Es wird von links aus geprüft, welche Zahlen kombiniert werden.

```
1 moveRowLeftHelper(listOf(2, 2, 2))
```

```
[4, 2]
```

Tipp: Zähle in einer `While`-Schleife durch einen Teil der Indizes der übergebenen Liste. Prüfe in jedem Schritt, ob das Element bei dem du gerade bist, mit dem nächsten Element übereinstimmt. In beiden Fällen müssen der Akkumulator und die Zählervariable unterschiedlich geändert werden.

1.9 Aufgabe

Implementiere eine Funktion `moveRowLeft`. Dieser wird eine Liste mit Integern übergeben. Du kannst davon ausgehen, dass diese nur 2-er Potenzen, die mindestens 2 sind, und 0-en enthält.

Die Funktion gibt die Liste zurück, die beim Kippen nach Links im Spiel 2048 entsteht. Diese Liste die zurückgegeben wird, ist genau so lang, wie die Liste die übergeben wird.

```
moveRowLeft(listOf(0, 0, 0, 2))
```

```
[2, 0, 0, 0]
```

```
moveRowLeft(listOf(0, 2, 0, 2, 0))
```

```
[4, 0, 0, 0, 0]
```

```
moveRowLeft(listOf(0, 2, 2, 4, 0))
```

```
[4, 4, 0, 0, 0]
```

```
moveRowLeft(listOf(0, 2, 2, 2))
```

```
[4, 2, 0, 0]
```

Hinweis: Nutze `removeZeros`, `moveRowLeftHelper` und `fill`!

1.10 Aufgabe

Implementiere eine Funktion `moveBoardLeft`. Dieser wird eine 2D-Liste mit Integern übergeben. Du kannst davon ausgehen, dass diese nur 2-er Potenzen, die mindestens 2 sind, und 0-en enthält.

Die Funktion gibt die $2D$-Liste zurück, die beim Kippen nach Links im Spiel 2048 entsteht.

```
moveBoardLeft(listOf(listOf(2, 2),
                     listOf(0, 4)))
```

```
[[4, 0], [4, 0]]
```

```
moveBoardLeft(listOf(listOf(0, 2, 4),
                     listOf(2, 2, 2),
                     listOf(2, 4, 2)))
```

```
[[2, 4, 0], [4, 2, 0], [2, 4, 2]]
```

Hinweis: Nutze `moveRowLeft`!

1.11 Aufgabe

Implementiere eine Funktion `getColumn`. Dieser wird eine 2D-Liste mit Integern übergeben. Du kannst davon ausgehen, dass alle Zeilen gleich lang sind. Außerdem wird eine Zahl i übergeben. Die Funktion gibt die i-te Spalte der $2D$-Liste zurück.

```
getColumn(listOf(listOf(2, 2),
                 listOf(0, 4)), 1)
```

```
[2, 4]
```

```
getColumn(listOf(listOf(1, 2, 3),
                 listOf(4, 5, 6),
                 listOf(7, 8, 9)),
                 2)
```

```
[3, 6, 9]
```

1.12 Aufgabe

Implementiere eine Funktion `reverse`. Dieser wird eine Liste mit Integern übergeben. Sie gibt eine Liste zurück in der die Reihenfolge der Elemente genau anders rum ist. Du kannst davon ausgehen, dass alle Zeilen gleich lang sind.

```
reverse(listOf(2, 1))
```

```
[1, 2]
```

```
reverse(listOf(2, 1, 0, -1, 70))
```

```
[70, -1, 0, 1, 2]
```

1.13 Aufgabe

Implementiere eine Funktion `rotateRight`. Dieser wird eine 2D-Liste mit Integern übergeben. Sie gibt die nach rechts gedrehte Liste zurück. Du kannst davon ausgehen, dass alle Zeilen gleich lang sind.

```
rotateRight(listOf(listOf(1, 2, 3),
                   listOf(4, 5, 6),
                   listOf(7, 8, 9)))
```

```
[[7, 4, 1], [8, 5, 2], [9, 6, 3]]
```

```
rotateRight(listOf(listOf(2, 2),
                   listOf(0, 4)))
```

```
[[0, 2], [4, 2]]
```

Hinweis: Nutze `getColumn` und `reverse`!

1.14 Aufgabe

Implementiere eine Funktion `rotateRight`. Dieser wird eine 2D-Liste mit Integern und ein Integer n übergeben. Sie gibt die n-mal nach rechts gedrehte Liste zurück. Du kannst davon ausgehen, dass alle Zeilen gleich lang sind.

```
rotateRight(listOf(listOf(1, 2, 3),
                   listOf(4, 5, 6),
                   listOf(7, 8, 9)), 2)
```

```
[[9, 8, 7], [6, 5, 4], [3, 2, 1]]
```

```
rotateRight(listOf(listOf(2, 2),
                   listOf(0, 4)), 3)
```

```
[[2, 4], [2, 0]]
```

Hinweis: Nutze `rotateRight`!

1.15 Aufgabe

Implementiere eine Funktion `rotateMoveLeftRotateBack`. Dieser wird eine 2D-Liste mit Integern und ein Integer n übergeben. Sie gibt die Liste zurück, nachdem diese n mal gedreht, einmal nach links gekippt, und wieder zurück gedreht wurde. Du kannst davon ausgehen, dass n kleiner als 4 ist.

```
rotateMoveLeftRotateBack(listOf(listOf(0, 2, 4),
                                listOf(2, 2, 2),
                                listOf(2, 4, 2)), 2)
```

```
[[0, 2, 4], [0, 2, 4], [2, 4, 2]]
```

```
rotateMoveLeftRotateBack(listOf(listOf(2, 2),
                                listOf(0, 4)), 3)
```

```
[[2, 2], [0, 4]]
```

Hinweis: Nutze `rotateRight` und `moveBoardLeft`!

1.16 Aufgabe

Anstatt das Kippen in jede Richtung getrennt zu implementieren, wollen wir nach der Eingabe einer Richtung, das Spielfeld so oft drehen, dass wir es nach links kippen können. Anschließend wird es wieder zurück gedreht.

Wenn wir z.B. das Spielfeld

```
val b = listOf(listOf(8, 2),
               listOf(16, 2))
```

nach unten kippen wollen, können wir dieses

- einmal nach rechts rotieren:

```
val b2 = rotateRight(b, 1)
```

```
printBoard(b2)
```

```
|     16 |      8 |
|      2 |      2 |
```

- nach links kippen:

```
val b3 = moveBoardLeft(b2)
```

```
printBoard(b3)
```

```
|     16 |      8 |
|      4 |      0 |
```

und anschließend wieder zurück drehen.

```
val b4 = rotateRight(b3, 3)
```

```
printBoard(b4)
```

```
|      8 |      0 |
|     16 |      4 |
```

Implementiere eine Funktion `computeNecessaryRotationsBeforeMove`. Dieser wird ein Character, der für eine Richtung steht(`w`, `a`, `s` oder `d`) übergeben. Sie berechnet, wie oft das Spielfeld vor dem Kippen gedreht werden muss.

```
computeNecessaryRotationsBeforeMove('s')
```

```
1
```

1.17 Aufgabe

Implementiere eine Funktion `move`. Dieser wird das Spielfeld und ein Character, der für eine Richtung steht(`w`, `a`, `s` oder `d`) übergeben. Sie gibt das Spielfeld nach dem Kippen in diese Richtung zurück.

```
move(listOf(listOf(8, 2),
            listOf(16, 2)),'s')
```

```
[[8, 0], [16, 4]]
```

```

move(listOf(listOf(0, 2, 4),
            listOf(2, 2, 2),
            listOf(2, 4, 2)), 'd')
```

```
[[0, 2, 4], [0, 2, 4], [2, 4, 2]]
```

Hinweis: Nutze `computeNecessaryRotationsBeforeMove` und `rotateMoveLeftRotateBack`!

1.18 Aufgabe

Implementiere eine Funktion `checkMovesPossible`. Dieser wird das Spielfeld übergeben. Sie gibt zurück, ob das Spielfeld beim Kippen in eine Richtung verändert wird.

```
checkMovesPossible(listOf(listOf(8, 2),
                          listOf(16, 2)))
```

```
true
```

```

checkMovesPossible(listOf(listOf(2, 4, 8),
                          listOf(4, 8, 4),
                          listOf(8, 4, 2)))
```

```
false
```

Hinweis: Nutze `rotateMoveLeftRotateBack`!

1.19 Aufgabe

Implementiere eine Funktion `contains`. Dieser wird eine Liste von Integern und ein einzelnes Integer übergeben. Sie gibt zurück, ob dieses Integer in der Liste enthalten ist.

```
contains(listOf(1, 2, 3) , 4)
```

```
false
```

```
contains(listOf(7, -3) , -3)
```

```
true
```

1.20 Aufgabe

Implementiere eine Funktion `flatten`. Dieser wird eine 2D-Liste von Integern übergeben. Sie gibt eine Liste zurück, in der alle Integer aus der 2D-Liste enthalten sind.

```
flatten(listOf(listOf(1, 2), listOf(3, 4), listOf(5)))
```

```
[1, 2, 3, 4, 5]
```

```
flatten(listOf(listOf(100), listOf(-10, 20, 50), listOf(3, 5)))
```

```
[100, -10, 20, 50, 3, 5]
```

1.21 Aufgabe

Implementiere eine Funktion `contains2D`. Dieser wird eine 2D-Liste von Integern und ein einzelnes Integer übergeben. Sie gibt zurück, ob dieses Integer in der 2D-Liste enthalten ist.

```
contains2D(listOf(listOf(1, 2, 3),
                  listOf(5, 8)), 8)
```

```
true
```

```
contains2D(listOf(listOf(1, 2, 3),
                  listOf(5, 8),
                  listOf(9, 2, 3)), 7)
```

```
false
```

Hinweis: Nutze `flatten` und `contains`!

1.22 Aufgabe

Verallgemeinere die Funktion, die du bis jetzt für dieses Projekt geschrieben hast, mit Typvariablen! Beachte, dass nicht jede Funktion verallgemeinert werden kann.

1.23 Aufgabe

Implementiere eine Funktion `setAt`. Dieser wird eine Liste xs, ein Index i und ein weiteres Element e übergeben. Dieses hat den selben Typ wie die Elemente in xs. Sie gibt eine Liste zurück, in der das Element an der Stelle i durch e ersetzt wurde. Ansonsten stimmt diese Liste mit xs überein.

```
setAt(listOf(1, 2, 3) , 1, 9)
```

```
[1, 9, 3]
```

```
setAt(listOf(true, false, true, false), 2, false)
```

```
[true, false, false, false]
```

1.24 Aufgabe

Implementiere eine Funktion `setAt`. Dieser wird eine 2D-Liste xs, ein Zeilenindex i, ein Spaltenindex j und ein weiteres Element e übergeben. Dieses hat den selben Typ wie die Elemente in xs. Sie gibt eine 2D-Liste zurück, in der das Element in Zeile i und Spalte j durch e ersetzt wurde. Ansonsten stimmt diese Liste mit xs überein.

```
setAt(listOf(listOf(1, 2, 3),
             listOf(5, 8)), 1, 0, 75)
```

```
[[1, 2, 3], [75, 8]]
```

```
setAt(listOf(listOf('u', 'o'),
             listOf('a', 'e', 'i'),
             listOf('a', 'c', 'f', 'p')), 2, 1, 't')
```

```
[[u, o], [a, e, i], [a, t, f, p]]
```

Hinweis: Nutze `setAt`!

1.25 Aufgabe

Implementiere eine Funktion `randomIndex`. Dieser wird eine Liste übergeben. Sie gibt den Index eines zufälligen Elements in dieser Liste zurück.

```
randomIndex(listOf('u', 'o'))
```

```
1
```

```
randomIndex(listOf(1, 5, 7, 10 , 500, 30, 0, -9, 200))
```

```
6
```

Importiere dafür mit

```
import kotlin.random.Random.Default.nextInt
```

die Funktion `nextInt`. Dieser wird eine positive ganze Zahl n übergeben. Sie gibt eine zufällige natürliche Zahl zurück, die **echt kleiner** ist als n.

```
nextInt(20)
```

```
16
```

```
nextInt(20)
```

```
14
```

1.26 Aufgabe

Implementiere eine Funktion `generateTwoOrFour`. Diese gibt zu 90% eine 2 und zu 10% eine 4 zurück.

```
generateTwoOrFour()
```

```
2
```

```
generateTwoOrFour()
```

```
2
```

1.27 Aufgabe

Implementiere eine Funktion `addTwoOrFour`. Dieser wird eine 2D-Liste übergeben. Du kannst davon ausgehen, dass alle Zeilen in dieser Liste gleich lang sind. Die Funktion gibt eine neue 2D-Liste zurück. In der neuen Liste wurde eine zufällig ausgewählte 0 durch eine 2 oder eine 4 ersetzt. Die neue Zahl ist zu 90% eine 2 und zu 10% eine 4. Wenn es keine 0 auf dem Spielfeld gibt, wird die übergebene 2D-Liste unverändert zurückgegeben.

```
addTwoOrFour(listOf(listOf(0, 2),
                    listOf(5, 8)))
```

```
[[2, 2], [5, 8]]
```

```
addTwoOrFour(listOf(listOf(0, 2, 7),
                    listOf(5, 8, 9),
                    listOf(7, 2, 0)))
```

```
[[0, 2, 7], [5, 8, 9], [7, 2, 2]]
```

Hinweis: Nutze `contains2D`, `randomIndex`, `generateTwoOrFour` und `setAt`!

1.28 Aufgabe

Implementiere eine Funktion `update`. Dieser wird eine Spielfeld als 2D-Liste übergeben. Außerdem wird ein Character, der für eine Richtung steht(`w`, `a`, `s` oder `d`) übergeben. Sie berechnet, dass Spielfeld nach dem Kippen in diese Richtung und dem Hinzufügen von zwei Elementen wie bei `addTwoOrFour`.

```
update(listOf(listOf(0, 2, 4),
              listOf(2, 2, 2),
              listOf(2, 4, 2)), 'd')
```

```
[[0, 2, 4], [2, 2, 4], [2, 4, 2]]
```

1.29 Aufgabe

Implementiere eine Funktion `showResult`. Dieser wird übergeben, ob der Spieler gewonnen oder verloren hat. Sie gibt diese Information in einem String zurück.

```
showResult(true)
```

```
Du hast gewonnen
```

```
showResult(false)
```

```
Du hast verloren
```

1.30 Aufgabe

Implementiere eine Funktion `repeat2D`. Dieser erzeugt eine 2D-Liste, in der an jeder Stelle das gleiche Element steht. Ihr werden die gewünschte Zeilen und Spaltenanzahl, sowie das Element übergeben.

```
repeat2D(3, 2, 0)
```

```
[[0, 0], [0, 0], [0, 0]]
```

```
repeat2D(4, 1, false)
```

```
[[false], [false], [false], [false]]
```

Hinweis: Nutze `fill`!

1.31 Aufgabe

Implementiere eine Funktion `createStartBoard`. Dieser erzeugt zuerst eine 2D-Liste aus 0-en. Anschließend werden an zwei zufälligen Positionen eine 2 oder eine 4 platziert. Die Wahrscheinlichkeit für eine 2 ist 90%. Der Funktion werden die gewünschte Zeilen und Spaltenanzahl übergeben.

```
createStartBoard(3, 2)
```

```
[[0, 0], [2, 2], [0, 0]]
```

```
createStartBoard(4, 1)
```

```
[[0], [2], [0], [2]]
```

Hinweis: Nutze `repeat2D` und `addTwoOrFour`!

1.32 Aufgabe

Schreibe eine Funktion `readChar`. Diese gibt den ersten Buchstaben einer Eingabe des Benutzers als `Char` zurück.

1.33 Aufgabe

Verbinde die bereits geschriebenen Funktion zur Funktion `play2048`. Dieser Funktion werden die gewünschte Zeilen und Spaltenanzahl übergeben. Das Spiel kann mit der Taste `q` beendet werden. Wenn die Zahl 2048 auf dem Spielfeld ist, hat der Spieler gewonnen. Bei einem früheren Spielende verliert er das Spiel.

```
|      0 |      0 |      0 |      0 |
|      0 |      0 |      0 |      0 |
|      2 |      4 |      0 |      0 |
|      0 |      0 |      0 |      0 |
w
|      2 |      4 |      2 |      0 |
|      0 |      0 |      0 |      0 |
|      0 |      0 |      0 |      0 |
|      0 |      0 |      0 |      0 |
```

```
a
|      2 |      4 |      2 |      0 |
|      0 |      0 |      0 |      0 |
|      0 |      0 |      2 |      0 |
|      0 |      0 |      0 |      0 |
w
|      2 |      4 |      4 |      0 |
|      0 |      0 |      0 |      0 |
|      0 |      2 |      0 |      0 |
|      0 |      0 |      0 |      0 |
a
|      2 |      8 |      0 |      0 |
|      2 |      0 |      0 |      0 |
|      2 |      0 |      0 |      0 |
|      0 |      0 |      0 |      0 |
w
|      4 |      8 |      0 |      0 |
|      2 |      0 |      2 |      0 |
|      0 |      0 |      0 |      0 |
|      0 |      0 |      0 |      0 |
q
Du hast verloren
```

Hinweis: Nutze `createStartBoard`, `checkMovesPossible`, `printBoard`, `readChar`, `update`, `contains2D` und `showResult`!