Advanced Programming Exercises

University of Skövde

August 26, 2020

Rules

- Use only immutable values (or no value declarations, just parameters, if you want it more challenging), functions and recursion.
- Functions should be written so that they can be called as given in the examples.
- Math built-in functions are allowed e.g. math.sqrt, math.pow.
- Other built-in functions are not allowed unless stated in the given exercise.

1 Recursion

1.1 Sum

Write a function `sum` of the form ***Int → Int*** which computes the sum of all integers from 0 till the given number. Formally:

$$\sum_{i=0}^{n} i$$

```
sum(3)        // = 6
sum(10)        // = 55
```

1.2 Factorial

Write a function `factorial` of the form ***Int → Int***, which computes the factorial of the given value.

$$\prod_{i=1}^{n} i$$

```
factorial(3)        // = 6
factorial(5)        // = 120
factorial(7)        // = 5040
```

1.3 Digits

Write a function `digits` of the form ***Int → Int***, which computes the number of digits in the given number.

```
digits(5)         //= 1
digits(46)        // = 2
digits(3465)        // = 4
```

1.4 Sum of digits

Write a function `sumDigits` of the form ***Int*** → ***Int***, which computes the sum of digits in the given number.

```
sumDigits(5)          //= 5
sumDigits(46)         // = 10
sumDigits(3465)         // = 18
```

1.5 Greatest Common Divisor

Write a function `gcd` of the form **(*Int*, *Int*)** → ***Int***, which computes the greatest common divisor between the to given numbers.

```
gcd(10, 55)         // = 5
gcd(21, 9)         // = 3
gcd(11, 23)         // = 1
```

1.6 Fibonacci

Write a function `fibonacci` of the form ***Int*** → ***Int***, which computes the Fibonacci value that corresponds to the given position. The Fibonacci sequence is given by $F_0 = 1$, $F_1 = 1$ and the numbers following by:

$$F_n = F_{n-1} + F_{n-2}$$

```
fibonacci(4)         // = 3
fibonacci(7)         // = 13
fibonacci(12)         // = 144
```

1.7 Pascal triangle

Write a function `pascal` of the form **(*Int*, *Int*)** → ***Int***, which computes the Pascal value for the given row and column. In other words, the two parameters are the coordinates to a value in the pascal triangle, and the function should return the value of the given coordinates.

```
pascal(1, 1) // = 1
pascal(3, 2) // = 2
pascal(8, 4) // = 35
```

2 Recursion with lists

2.1 Sum elements

Write a function `sum` of the form ***List*[*Int*]** → ***Int***, which computes the sum of elements of the given list.

```
sum(List(1, 2, 3, 4, 5))         // = 15
sum(List(45, -3, 8, -23))         // = 27
```

2.2 Square elements

Write a function `square` of the form ***List*[*Int*]** → ***List*[*Int*]**, which squares the elements of the given list and returns them in a new list.

```
square(List(1, 2, 3, 4, 5))        // = List(1, 4, 9, 16, 25)
square(List(12, 56, 32))         // = List(144, 3136, 1024)
```

2.3 Maximum value

Write a function `max` of the form ***List*[*Int*] → *Int***, which returns the largest number in the given list.

```
max(List(1, 6, 3, 7, 2))        // = 7
max(List(-1, 56, 34, -43, 4, 110))        // =  110
max(List(5))        // = 5
```

2.4 Reverse

Write a function `reverse` of the form ***List*[*Int*] → *List*[*Int*]**, which returns (as a new list) the elements of the given list in reverse order.

```
reverse(List(1, 2, 3, 4))                // = List(4, 3, 2, 1)
reverse(List(0, 34, -43, 4, 110))        // =  List(110, 4, -43, 34, 0)
reverse(List(5))                        // = List(5)
```

2.5 Concatenation

Write a function `concatenate` of the form **(*List*[*Int*], *List*[*Int*]) → *List*[*Int*]**, which returns a new list with the values of both given lists.

```
concatenate(List(1, 2, 3), List(4, 5))        // = List(1, 2, 3, 4, 5)
concatenate(List(3), List(3, 4, 4))        // = List(3, 3, 4, 4)
```

2.6 Zip

Write a function `zip` of the form **(*List*[*Int*], *List*[*Int*]) → *List*[(*Int*, *Int*)]**, which takes two lists and returns a new one where elements are tuples containing values from both lists.

```
zip(List(1, 2, 3), List(6, 3, 4))        // = List((1, 6), (2, 3), (3, 4))
zip(List(1, 2), List(6, 3, 4))        // = List((1, 6), (2, 3))
zip(List(1, 2, 3), List(6))        // = List((1, 6))
```

2.7 Has subsequence

Write a function `hasSubsequence` ' which checks whether a list contains another list.

```
val l = 1 to 10 toList
hasSubsequence(l, List(1, 2))   // = true
hasSubsequence(l, List(5, 6, 7)) // = true
hasSubsequence(l, List(7, 8, 10))   // = false
```

2.8 N^{th} smallest number

Write a function `nSmallest` which takes a list of integers and a number, and returns the number in the list which is the N^{th} smallest.

```
val ns = List(3, 7, 1, 9, 3, 5, 8)
nSmallest(ns, 2)        // = 3 (second smallest)
nSmallest(ns, 4)        // = 5 (fourth smallest)
```

2.9 Standard deviation

Write a function `std` which computes the standard deviation for a given list. Recall that the standard deviation is defined as follows:

$$\sqrt{\frac{\sum_{i=0}^{n}(x_i - \bar{x})^2}{N-1}}$$

Where $\bar{x}$ is the mean of all values in the list.

```
std(List(10, 12, 23, 5, 16)) // = 6.0464...
```

2.10 L-Norm

The norm of a vector is a positive value representing its length. In this case, it is not the length as in the number of elements, but rather the size in terms of its values. There are different norms depending on how you calculate the length. The following are two norm examples:

$$L^1 = \sum_{i=0}^{n} |x_i|, \quad L^2 = \sqrt{\sum_{i=0}^{n} x_i^2}$$

The former is also known as Manhattan or Taxicab, and the latter as Euclidean (a generalization of these two is referred as p-norm, see Exercise 4.4). An example given the following one, two and three dimensional vectors:

$$a = \begin{bmatrix} 2 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ -3 \end{bmatrix}, \quad c = \begin{bmatrix} 2 \\ -3 \\ 5 \end{bmatrix}$$

Their norms would be $L^1(a) = 2$, $L^1(b) = 5$, $L^1(c) = 10$, and $L^2(a) = 2$, $L^2(b) = 3.61$, $L^2(c) = 6.16$. Write a function `l1` and a function `l2`, each taking a list of `Double`, and returning their correspoding norm.

```
l1(List(2.0, -3.0, 1.0))    // = 10
l2(List(2.0, -3.0, 1.0))        // = 6.1644..
```

2.11 Dot product

The dot product is an algebraic operation that, given two vectors a and b returns a the sum of the products their elements. Formally:

$$\sum_{i=0}^{n} a_i b_i$$

Write a function `dot` which takes two lists of integers and returns their dot product. The function should be tail recursive.

```
dot(List(1, 1, 1), List(1, 1, 1))        // = 3
dot(List(1, 2, 3, 4), List(9, 8, 7, 6))        // = 70
```

2.12 Standarization / Standard score / Z-Score

The z-score (also called standard score or standarization value) is a measure that tells how many standard deviations a value is from the mean (i.e., how far the value is from the mean). Formally:

$$z_i = \frac{x_i - \bar{x}}{\sigma}$$

Where z_i is the z-score of a value i in vector (list) x, while $\bar{x}$ is the mean of the values in the vector and σ the standard deviation. Write function `standarize` which takes a list of doubles, their mean and standard deviation, and returns a list with the z-scores for each element. Note: You may use `math.sqrt()` to compute the square root of a number.

```
val l = List(23.2, 65.5, 29.7, 65.5, 83.2, 13.9, 50.8, 1.0)
val u = mean(l)
val s = std(l)
standarize(l, u, s)        // = List(-0.636.., 0.826.., ..)
```

2.13 Pearson correlation

The Pearson correlation between two variables will tell us how correlated they are, e.g., given to variables a and b: if b increases when a increases (or vice versa), then they are positively correlated; if, on the other hand, b decreases when a increases, then they are negatively correlated; finally, if neither case takes place, we may say that there is no correlation. Formally, Pearson correlation is defined as follows:

$$pearson(a, b) = \frac{\sum_{i=1}^{n}(a_i - \bar{a})(b_i - \bar{b})}{\sqrt{\sum_{i=1}^{n}(a_i - \bar{a})^2}\sqrt{\sum_{i=1}^{n}(b_i - \bar{b})^2}}$$

Where a and b are vectors, and $\bar{a}$ and $\bar{b}$ are their corresponding means. Write a function `pearson` which takes two lists of integers and returns their correlation value. Note: You may use `math.sqrt()` to compute the square root of a number.

```
val a = 1 to 10 toList
val b = (1 to 5 toList) ++: (1 to 5 toList).reverse
pearson(a, a)         // = 1.0
pearson(a, a.reverse) // = -1.0
pearson(a, b)             // = 0.0
```

3 Higher-order functions

3.1 Count

Create a function `count` which takes a list of integers, and a function of the form $Int \to Boolean$; and returns the count of those elements in the list which comply with the given function.

```
map(List(1, 2, 3, 4, 5), x => x % 2 == 0)        // = 2
map(List(1, 2, 3, 4, 5), x => x % 2 != 0)        // = 3
```

3.2 Map

Create a function `map` which takes a list of integers, and a function of the form $Int \to Int$; and returns a new list with values transformed by the given function.

```
map(List(1, 2, 3, 4, 5), x => x * x)        // = List(1, 4, 9, 16, 25)
map(List(1, 2, 3, 4, 5), x => x * x * x)        // = List(1, 8, 27, 64, 125)
```

3.3 Sorted

Write a function `sorted` which checks whether a list of integers is sorted according to a comparison function of the form $(Int, Int) \to Boolean$.

```
sorted(List(1, 2, 3, 4, 5), (a, b) => a < b)        // = true
sorted(List(1, 2, 3, 4, 5, 3), (a, b) => a < b)       // = false
```

3.4 Filter

Write a function `filter` which takes a list of integers and a function, and returns a new list with only those values which evaluate to True according to the given function.

```
filter(List(1, 2, 3, 4, 5), x => x % 2 == 0)        // = List(2, 4)
filter(List(9, 8, 7, 6, 5, 4), x => x % 3 == 0)       // = List(9, 6)
filter(List(60, 23, 11, 76, 42, 9), x => x > 50)       // = List(60, 76)
```

3.5 Partition

Write a function `partition` which takes a list of integers and a function of the form $Int \to Boolean$; and returns a tuple with to lists, one of elements that evaluate to true (as given by the function), and those to false.

```
partition(List(1, 2, 3, 4), x => x % 2 == 0) // = (List(2, 4), List(1, 3))
partition(List(9, 8, 7, 6, 5), x => x > 6) // = (List(9, 8), List(6, 5))
```

3.6 Reduce

Write a function `reduce` which takes a list of integers and a function, and returns an integer resulting from having combined all values of the list using the given function.

```
reduce(List(1, 8, 4, 3, 9, 5), _ + _)        // = 30.
reduce(List(23, 76, 34, 84, 24, 58), (a, b) => a.max(b))       // = 84.
```

3.7 Fold

Write a function `fold` which takes an initial value, a list of integers and a function, and returns an integer resulting from having combined all values of the list, starting with the initial value, using the given function.

```
fold(0, List(1, 8, 4, 3, 9, 5), _ + _)        // = 30.
fold(1, List(1, 8, 4, 3, 9, 5), _ * _)        // = 4320.
```

3.8 Merge and reduce

Write a higher-order function `mergeReduce` which takes two lists of integers and two functions, `merge` and `reduce`, and returns an integer. The idea is to "merge" both lists together into one, and then "reduce" the values of the resulting list to a single integer.
For example, take the following vectors

$$a = [3, 7, 2, 9], \quad b = [1, 8, 4, 6]$$

Merging a and b using a product function p would look as follows

$$[p(3,1), p(7,8), p(2,4), p(9,6)] = [3, 56, 8, 54]$$

Reducing the resulting list with an addition function s would then be

$$s(3, s(56, s(8, 54)) = 121$$

The function `mergeReduce` can be seen as a generalization of functions like norm and the dot product (see Exercises 2.10 and 2.11).

```
val a = List(3, 7, 2, 9)
val b = List(1, 8, 4, 6)
val prod = (x: Int, y: Int) => x * y     // merge function
val sum = (x: Int, y: Int) => x + y      // reduce function
mergeReduce(a, b, prod, sum) // = 121
```

3.9 Bisection method

This is a method that allow us to find (or estimate) the root of a continuous function. That is, given a function ff, the bisection method will find the value for x such that $ff(x) = 0$. (More on root functions here).

Write a function `bisection` with four parameters: `low`, `high`, `epsilon` and `ff`. Here, `low` and `high` are doubles that define an interval where we know the zero is found, i.e.:

$$ff(low) < 0 < ff(high)$$

`epsilon`, on the other hand, is a positive double that defines the precision of the estimation, i.e., how close to zero. The smaller the value, the closer to zero. Finally, `ff` is the function for which we are looking the root value. This should be given in the form `Double => Double`.

```
val ffx2n1: (Double => Double) = (x) => {(x*x)-1}
val ffx2n2: (Double => Double) = (x) => {(x*x)-2}
bisection (-2.0,2.0,0.0001,ffx2n1)  // = -1.0
bisection (-2.0,2.0,0.0001,ffx2n2)  // = -1.4141..
bisection (0.0,2.0,0.0001,ffx2n2)   // = 1.4141..
```

4 Currification

4.1 Currified function

Write a function `currified` which takes a three doubles and computes the following:

$$a * (b + c)^2$$

The function should be written as a function literal – i.e. using `val` and not `def` – and written in a currified manner.

```
val f1 = currified(5)
val f2 = f1(3)
f2(4)       // = 245
currified(5)(3)(4)  // = 245
```

4.2 Currified filter

Write a currified version of the `filter` function but, in this case, the evaluation function parameter is given first and the list second.

```
val l = List(1, 2, 3, 4, 5, 6, 7)
val gt = filter(x => x > 4)
val evens = filter(x => x % 2 == 0)
gt(l)    // = List(5, 6, 7)
evens(l) // = list(2, 4, 6)
```

4.3 Scale

Write a function "scale" which scales the value of a Double to a given range. The general formula for scaling is as follows:

$$f(min, max, a, b, x) = \frac{(b-a)(x-min)}{max-min} + a$$

where min and max correspond to the minimum and maximum values that your original data can take, while a and b denote the target range to which you want to scale x.

```
val a: List[Double] = List(23, 73, 5, 43, 18, 84, 3)
scale(a.min, a.max)(0, 1)(3)    // = 0.0
scale(a.min, a.max)(0, 1)(84)   // = 1.0

val s = scale(a.min, a.max)_
s(0, 1)(a.head)     // = 0.246913...
s(10, 20)(a.head)   // = 12.46913...

a.map(s(0, 10)) // = List(2.469, 8.641, 0.246, 4.938, 1.851, 10.0, 0.0)
```

Write it as a `def` function and then as a function literal (with `val`).

4.4 Minkowski distance

The Minkowski distance is a measure that tells how far apart are two data points in the space. It is a generalization of the Euclidean and Manhattan distance. Write a currified function "minkowski" which takes a double and two lists of doubles, and returns their distance in terms of the Minkowski distance. The distance is given as follows:

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

for $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ and where $|\cdot|$ is the absolute value, as usual.

```
val a = List(1.0, 2.0, 3.0, 4.0, 5.0)
val b = List(1.0, 2.0, 1.0, 2.0, 5.0)
val euclidean = minkowski(2)
val manhattan = minkowski(1)
minkowski(1.5)(a, b)        // = 3.1748021039363987
euclidean(a, b)         // = 2.8284271247461903
manhattan(a, b)         // = 4.0
```

5 Streams

5.1 Natural numbers

Write a function `numbersFrom` which takes an integer and returns a Stream with the natural numbers starting from the given number.

```
numbersFrom(5).take(8).toList        // = List(5, 6, 7, 8, 9, 10, 11, 12)
numbersFrom(11).take(8).toList        // = List(11, 12, 13, 14, 15, 16, 17, 18)
```

5.2 Fibonacci stream

Write a function `fibonacciStream` with no parameters which returns a Stream with the Fibonacci sequence.

```
fibonacciStream.take(8).toList        // = List(1, 1, 2, 3, 5, 8, 13, 21)
```

5.3 Moving average

The moving average is a method for smoothing values in a time series. It is a way to reduce radical changes seen in a (possible infinite) list of values. Write a function "averages" which takes an integer and a stream of doubles, and returns a Stream with the moving averages. The integer represents the order of the moving average.

A moving average of order 3 can be defined as follows:

$$\bar{x}_i = \frac{1}{3}(x_{i-1} + x_i + x_{i+1})$$

A moving average of order 5 would then be:

$$\bar{x}_i = \frac{1}{5}(x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2})$$

Note: You may use Stream built-in functions `take` and `sum`.

```
val ns = numbersFrom(0)
averages(3, ns).take(3).toList        // = List(4.7, 5.1, 4.466666666666667)
averages(5, ns).take(3).toList        // = List(4.38, 5.24, 5.42)
```

5.4 Leibniz sequence for Pi

Write a function `leibnizStream` with no parameters, which returns a Stream with the values for the Leibniz sequence as given by:

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
leibnizStream().take(100).sum * 4        // = 3.1315929035585537
leibnizStream().take(1000).sum * 4        // = 3.140592653839794
```

We can then make use of the sequence to compute an estimate of Pi.

5.5 Secant method

The secant method is another method to find/estimate the root of a function (Exercise 3.9 uses the bisection method). It is based on the following recurrence relation:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

Write a function `secantStream` which solves this problem in terms of a stream (infinite structure) of all x_i, starting from a given x_0 and x_1, with a given function f (with signature `Double => Double`). The root of the function f will then be given by the first value of the stream which satisfies:

$$|x_n - x_{n-1}| \leq \epsilon |x_n|$$

This is the convergence criterion, where ϵ (epsilon) is a positive double defining the precision of the estimation (as in the bisection method). Write a another function "secantMethod" that, given a stream (computed with `secantStream`), and a value ϵ (epsilon), returns the first x_n that fulfills the criterion. Note: you may use built-in functions for "secantMethod".

```
val f2m8: (Double => Double) = (x) => { x*x - 8.0 }
val s = secantStream(0,10,f2m8) // = Stream(0.799.., ?)
secantMethod(s, 0.00001)    // = 2.828427110105244
```

5.6 Sieve of Eratosthenes

Write a function `primesStream` with no parameters which returns a Stream of only prime numbers. Note: Built-in functions allowed.

```
primesStream.take(10).toList // = List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

5.7 Streams generalized

The following function signature provides a general way to create streams:

```
def unfold[A, B](s: B, f: B => Option[(A, B)]): Stream[A] = ???
```

The `s` parameter represents an initial state. The `f` parameter, on the other hand, is a function that takes a state and returns an `Option` element with a tuple of two values: an element of the stream, and the following state with which the next element of the stream is to be computed. If the `f` function produces `None`, then the stream terminates.

Write the body of the function `unfold` and then make a call to it such that a stream of natural numbers (1, 2, 3, ...) is produced.

6 More exercises

6.1 Balanced

Write a function `balanced` which takes a string and returns whether parentheses are balanced or not. The string parameter can be dealt as a List of chars.

```
balanced("hello (world)".toList)        // = true (balanced).
balanced("(()".toList)        // = false (unbalanced).
balanced("()(())".toList)        // = true.
balanced("(a)(b(c)()".toList)        // = false.
```

6.2 Combinations of 0s and 1s

Write a function `combinations` which takes an integer, and returns all possible combinations of lists of 0s and 1s with length equal to the given number.

```
combinations(2) // = List(List(0, 0), List(0, 1), List(1, 0), List(1, 1))
combinations(3) // = List(List(0, 0, 0), List(0, 0, 1), List(0, 1, 0),
// List(0, 1, 1), List(1, 0, 0), List(1, 0, 1), List(1, 1, 0), List(1, 1, 1))
```

6.3 Count change

Write a function `countChange` which counts how many different ways you can make change for an amount, given a list of coin denominations. For example, there are 3 ways to give change for 4 if you have coins with denomination 1 and 2: 1+1+1+1, 1+1+2, 2+2.

```
countChange(4, List(1, 2))        // = 3
countChange(12, List(2, 3, 4))        // = 7
```

6.4 Quicksort

Write a function `quicksort` which takes a list of integers and returns its sorted version. See https://en.wikipedia.org/wiki/Quicksort. Note: you may use the built-in function `filter`.

```
quicksort(List(1, 3, 2, 5, 4))        // = List(1, 2, 3, 4, 5)
```

6.5 Dissimilarity/distance matrix

Computing the distance (similarity) between each element of a list (or a dataset for that matter) results in a so called dissimilarity matrix. For example, given a vector $x = [-3, 6, 2]$, and a distance function $d(a, b) = |a - b|$ (Manhattan), its dissimilarity matrix is as follows:

	-3	6	2
-3	$d(-3,-3)$	$d(-3,6)$	$d(-3,2)$
6	$d(6,-3)$	$d(6,6)$	$d(6,2)$
2	$d(-2,-3)$	$d(2,6)$	$d(2,2)$

$=>$

	-3	6	2
-3	0	9	5
6	9	0	8
2	5	8	0

Observe that the elements in the diagonal cells are all zero, because the distance from one element to itself is always zero. This is an example of a one-dimensional dataset, yet the logic can be extended to handle n-dimensional datasets, e.g.:

	$[-3, 2]$	$[6, 4]$	$[2, -1]$
$[-3, 2]$	d([-3, 2],[-3, 2])	d([-3, 2],[6,4])	d([-3, 2],[2, -1])
$[6, 4]$	d([6,4],[-3, 2])	d([6,4],[6,4])	d([6,4],[2, -1])
$[2, -1]$	d([2, -1],[-3, 2])	d([2, -1],[6,4])	d([2, -1], 2)

The distance function d, in this case, computes the distance between two vectors (see Exercise 4.4). Write a function `distance` which takes a dataset in the form of `List[List[Double]]`, and returns an element of the same type, where each inner list corresponds to a row of the dissimilarity matrix. Use Manhattan distance.

```
val dataset = List(List(-3.0, 2.0), List(6.0, 4.0), List(2.0, -1.0))
distance(dataset)
```

6.6 Transpose

A matrix in linear algebra

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Can be transposed by making its columns into rows

$$A^t = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

A row in Scala can be defined as a list of numbers, and a matrix as a list of rows. We can define the previous using the following type aliases:

```
type Row = List[Int]
type Matrix = List[Row]
```

Write a function `transpose` which takes a `Matrix` and returns its transposed version.

```
val m: Matrix = List(1 :: 2 :: Nil, 3 :: 4 :: Nil, 5 :: 6 :: Nil)
val mt: Matrix = transpose(m)
m.map(_.mkString(" ")).foreach(println)
// 1 2
// 3 4
// 5 6
mt.map(_.mkString(" ")).foreach(println)
// 1 3 5
// 2 4 6
```

6.7 Matrix multiplication

An example of matrix multiplication is the following

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}, AB = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

Write a function `multiply` which takes two matrices and returns their multiplication.

```
val a: Matrix = List(1 :: 2 :: 3 :: Nil, 4 :: 5 :: 6 :: Nil)
val b: Matrix = List(7 :: 8 :: Nil, 9 :: 10 :: Nil, 11 :: 12 :: Nil)
val ab: Matrix = multiply(a, b)
ab.map(_.mkString(" ")).foreach(println)
// 58 64
// 139 154
```

6.8 15-Puzzle

This is a tile-sliding problem called 15-pluzzle which you should solve using search algorithms. In this case, however, you will work with the simplified version, the 8-puzzle, which can be represented by a 3x3 matrix, where one cell/tile is "empty" and the rest are filled with numbers from 1 to 8, e.g.:

	1	3
4	2	5
6	7	8

The goal of the puzzle is to use the empty space to move neighboring tiles until the entire matrix is sorted, e.g.:

	1	3
4	2	5
6	7	8

1		3
4	2	5
6	7	8

1	2	3
4		5
6	7	8

For the purpose of this exercise, we do not care where the empty space ends. What matters is that the numbered tiles are sorted in ascending manner from left to right, top to bottom. It is important to note that this problem is not solvable for half of the possible initial configurations.

In this Github project you will find an implementation of the 8-puzzle as defined here. Your search solutions are to be coded in the `Search.scala` file. Once you have coded your solutions you can test them through the `Main.scala`. You will need sbt to run your solution.