Main Part 4 (Scala, 11 Marks)

> *"The problem with object-oriented languages is they've got all this implicit, environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."*
> *— Joe Armstrong (creator of the Erlang programming language)*

This part is about searching and backtracking. You are asked to implement Scala programs that solve various versions of the *Knight's Tour Problem* on a chessboard.

Important

- Make sure the files you submit can be processed by just calling `scala <<filename.scala>>` on the command line.[1] Use the template files provided and do not make any changes to arguments of functions or to any types. You are free to implement any auxiliary function you might need.
- **Do not leave any test cases running in your code because this might slow down your program!** Comment out test cases before submission, otherwise you might hit a time-out.
- Do not use any mutable data structures in your submissions! They are not needed. This means you cannot create new `Arrays` or `ListBuffers`, for example.
- Do not use `return` in your code! It has a different meaning in Scala than in Java. It changes the meaning of your program, and you should never use it.
- Do not use `var`! This declares a mutable variable. Only use `val`!
- Do not use any parallel collections! No `.par` therefore! Our testing and marking infrastructure is not set up for it.

Also note that the running time of each part will be restricted to a maximum of 30 seconds on my laptop: If you calculate a result once, try to avoid to calculate the result again. Feel free to copy any code you need from files `knight1.scala`, `knight2.scala` and `knight3.scala`.

Disclaimer

It should be understood that the work you submit represents your **own** effort! You have implemented the code entirely on your own. You have not copied

[1] All major OSes, including Windows, have a command line. So there is no good reason to not download Scala, install it and run it on your own computer. Just do it!

from anyone else. Do not be tempted to ask Github Copilot for help or do any other shenanigans like this! An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

Background

The *Knight's Tour Problem* is about finding a tour such that the knight visits every field on an $n \times n$ chessboard once. For example on a 5×5 chessboard, a knight's tour is:

4	24	11	6	17	0
3	19	16	23	12	7
2	10	5	18	1	22
1	15	20	3	8	13
0	4	9	14	21	2
	0	1	2	3	4

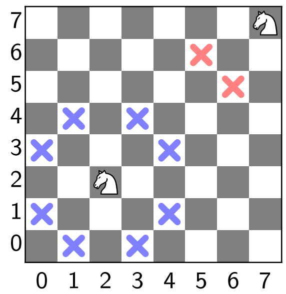
This tour starts in the right-upper corner, then moves to field $(3, 2)$, then $(4, 0)$ and so on. There are no knight's tours on 2×2, 3×3 and 4×4 chessboards, but for every bigger board there is.

A knight's tour is called closed, if the last step in the tour is within a knight's move to the beginning of the tour. So the above knight's tour is not closed because the last step on field $(0, 4)$ is not within the reach of the first step on $(4, 4)$. It turns out there is no closed knight's tour on a 5×5 board. But there are on a 6×6 board and on bigger ones, for example

10	5	18	25	16	7
31	26	9	6	19	24
4	11	30	17	8	15
29	32	27	0	23	20
12	3	34	21	14	1
33	28	13	2	35	22

where the 35th move can join up again with the 0th move.

If you cannot remember how a knight moves in chess, or never played chess, below are all potential moves indicated for two knights, one on field $(2, 2)$ (blue moves) and another on $(7, 7)$ (red moves):



Reference Implementation

This Scala part comes with three reference implementations in form of `jar`-files. This allows you to run any test cases on your own computer. For example you can call Scala on the command line with the option `-cp knight1.jar` and then query any function from the `knight1.scala` template file. As usual you have to prefix the calls with `M4a`, `M4b`, `M4c` and `M4d`. Since some of the calls are time sensitive, I included some timing information. For example

```
$ scala -cp knight1.jar
scala> M4a.enum_tours(5, List((0, 0))).length
Time needed: 1.722 secs.
res0: Int = 304

scala> M4a.print_board(8, M4a.first_tour(8, List((0, 0))).get)
Time needed: 15.411 secs.

 51  46  55  44  53   4  21  12
 56  43  52   3  22  13  24   5
 47  50  45  54  25  20  11  14
 42  57   2  49  40  23   6  19
 35  48  41  26  61  10  15  28
 58   1  36  39  32  27  18   7
 37  34  31  60   9  62  29  16
  0  59  38  33  30  17   8  63
```

Hints

Useful list functions: `.contains(..)` checks whether an element is in a list, `.flatten` turns a list of lists into just a list, `_::_` puts an element on the head of the list, `.head` gives you the first element of a list (make sure the list is not `Nil`); a useful option function: `.isDefined` returns true, if an option is `Some(..)`; anonymous functions can be constructed using `(x:Int) => ...`, this function takes an `Int` as an argument; a useful list function: `.sortBy` sorts a list according to a component given by the function; a function can be tested to be tail-recursive by annotation `@tailrec`, which is made available by importing `scala.annotation.tailrec`.

Tasks

You are asked to implement the knight's tour problem such that the dimension of the board can be changed. Therefore most functions will take the dimension of the board as an argument. The fun with this problem is that even for small chessboard dimensions it has already an incredibly large search space—finding a tour is like finding a needle in a haystack. In the first task we want to see how far we get with exhaustively exploring the complete search space for small chessboards.

Let us first fix the basic datastructures for the implementation. The board dimension is an integer. A position (or field) on the chessboard is a pair of integers, like $(0,0)$. A path is a list of positions. The first (or 0th move) in a path is the last element in this list; and the last move in the path is the first element. For example the path for the 5×5 chessboard above is represented by

$$\texttt{List(}\underbrace{\texttt{(0, 4)}}_{24}\texttt{, }\underbrace{\texttt{(2, 3)}}_{23}\texttt{, ..., }\underbrace{\texttt{(3, 2)}}_{1}\texttt{, }\underbrace{\texttt{(4, 4)}}_{0}\texttt{)}$$

Suppose the dimension of a chessboard is n, then a path is a tour if the length of the path is $n \times n$, each element occurs only once in the path, and each move follows the rules of how a knight moves (see above for the rules).

Task 1 (file knight1.scala)

(1) Implement an `is_legal` function that takes a dimension, a path and a position as arguments and tests whether the position is inside the board and not yet element in the path. [1 Mark]

(2) Implement a `legal_moves` function that calculates for a position all legal onward moves. If the onward moves are placed on a circle, you should produce them starting from "12-o'clock" following in clockwise order. For example on an 8×8 board for a knight at position $(2,2)$ and otherwise empty board, the legal-moves function should produce the onward positions in this order:

```
List((3,4), (4,3), (4,1), (3,0), (1,0), (0,1), (0,3), (1,4))
```

If the board is not empty, then maybe some of the moves need to be filtered out from this list. For a knight on field $(7,7)$ and an empty board, the legal moves are

```
List((6,5), (5,6))
```

[1 Mark]

(3) Implement two recursive functions (`count_tours` and `enum_tours`). They each take a dimension and a path as arguments. They exhaustively search for tours starting from the given path. The first function counts all possible tours (there can be none for certain board sizes) and the second collects all tours in a list of paths. These functions will be called with a path containing a single position—the starting field. They are expected to extend this path so as to find all tours starting from the given position.

[1 Mark]

Test data: For the marking, the functions in (3) will be called with board sizes up to 5×5. If you search for tours on a 5×5 board starting only from field $(0, 0)$, there are 304 of tours. If you try out every field of a 5×5-board as a starting field and add up all tours, you obtain 1728. A 6×6 board is already too large to be searched exhaustively.[2]

(4) Implement a `first`-function. This function takes a list of positions and a function f as arguments; f is the name we give to this argument). The function f takes a position as argument and produces an optional path. So f's type is `Pos => Option[Path]`. The idea behind the `first`-function is as follows:

$$\begin{aligned} \mathit{first}(\texttt{Nil}, f) &\stackrel{\text{def}}{=} \texttt{None} \\ \mathit{first}(x :: xs, f) &\stackrel{\text{def}}{=} \begin{cases} f(x) & \text{if } f(x) \neq \texttt{None} \\ \mathit{first}(xs, f) & \text{otherwise} \end{cases} \end{aligned}$$

That is, we want to find the first position where the result of f is not `None`, if there is one. Note that 'inside' `first`, you do not (need to) know anything about the argument f except its type, namely `Pos => Option[Path]`. If you want to find out what the result of f is on a particular argument, say x, you can just write $f(x)$. There is one additional point however you should take into account when implementing `first`: you will need to calculate what the result of $f(x)$ is; your code should do this only **once** and for as **few** elements in the list as possible! Do not calculate $f(x)$ for all elements and then see which is the first `Some`.

[1 Mark]

(5) Implement a `first_tour` function that uses the `first`-function from (4), and searches recursively for single tour. As there might not be such a tour at all, the `first_tour` function needs to return a value of type `Option[Path]`.

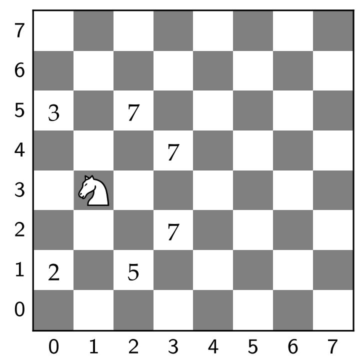
[1 Mark]

Testing: The `first_tour` function will be called with board sizes of up to 8×8.

[2] For your interest, the number of tours on 6×6, 7×7 and 8×8 are 6637920, 165575218320, 19591828170979904, respectively.

Task 2 (file knight2.scala)

As you should have seen in the earlier parts, a naive search for tours beyond 8×8 boards and also searching for closed tours even on small boards takes too much time. There is a heuristics, called Warnsdorf's Rule that can speed up finding a tour. This heuristics states that a knight is moved so that it always proceeds to the field from which the knight will have the fewest onward moves. For example for a knight on field $(1, 3)$, the field $(0, 1)$ has the fewest possible onward moves, namely 2.



Warnsdorf's Rule states that the moves on the board above should be tried in the order

$$(0, 1), (0, 5), (2, 1), (2, 5), (3, 4), (3, 2)$$

Whenever there are ties, the corresponding onward moves can be in any order. When calculating the number of onward moves for each field, we do not count moves that revisit any field already visited.

(6) Write a function `ordered_moves` that calculates a list of onward moves like in (2) but orders them according to Warnsdorf's Rule. That means moves with the fewest legal onward moves should come first (in order to be tried out first). [1 Mark]

(7) Implement a `first_closed_tour_heuristics` function that searches for a single **closed** tour on a 6×6 board. It should try out onward moves according to the `ordered_moves` function from (6). It is more likely to find a solution when started in the middle of the board (that is position $(dimension/2, dimension/2)$). [1 Mark]

(8) Implement a `first_tour_heuristics` function for boards up to 30×30. It is the same function as in (7) but searches for tours (not just closed tours). It might be called with any field on the board as starting field. [1 Mark]

Task 3 (file knight3.scala)

(9) Implement a function `tour_on_mega_board` which is the same function as in (8), **but** should be able to deal with boards up to 70×70 **within 30 seconds** (on my laptop). This will be tested by starting from field $(0, 0)$. You have to be careful to write a tail-recursive function otherwise you will get problems with stack-overflows. Please observe the requirements about the submissions: no tricks involving **.par**.

The timelimit of 30 seconds is with respect to the laptop on which the marking will happen. You can roughly estimate how well your implementation performs by running `knight3.jar` on your computer. For example the reference implementation shows on my laptop:

```
$ scala -cp knight3.jar

scala> M4c.tour_on_mega_board(70, List((0, 0)))
Time needed: 9.484 secs.
...<<long_list>>...
```

[1 Mark]

Task 4 (file knight4.scala)

(10) In this task we want to solve the problem of finding a single tour (if there exists one) on what is sometimes called "mutilated" chessboards, for example rectangular chessbords or chessboards where fields are missing. For this implement a search function

```
def one_tour_pred(dim: Int, path: Path,
                  n: Int, f: Pos => Boolean): Option[Path]
```

which has, like before, the dimension and current path as arguments, and in addtion it takes an integer, which specifies the length of the longest path (or length of the path after which the search should backtrack), and a function from positions to Booleans. This function acts as a predicate in order to restrict which onward legal moves should be considered in the search. The function `one_tour_pred` should return a single tour (`Some`-case), if one or more tours exist, and `None`, if no tour exists. For example when called with

```
one_tour_pred(7, List((0, 0)), 35, x => x._1 < 5)
```

we are looking for a tour starting from position `(0,0)` on a 7×5 board, where the maximum length of the path is 35. The predicate `x => x._1 < 5` ensures that no position with an x-coordinate bigger than 4 is considered. One possible solution is

```
  0   13   22   33   28   11   20
 23   32    1   12   21   34   27
 14    7   16   29    2   19   10
 31   24    5    8   17   26    3
  6   15   30   25    4    9   18
 -1   -1   -1   -1   -1   -1   -1
 -1   -1   -1   -1   -1   -1   -1
```

where `print_board` prints a `-1` for all fields that have not been visited.

[2 Marks]