Macquarie University Dept of Computing

COMP3000 Programming Languages 2020

Assignment 1 - Snake

Due: see submission page
Worth: 10% of unit assessment

Marks breakdown:

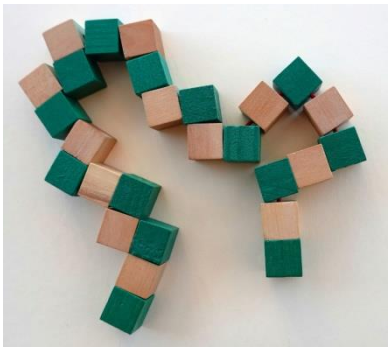
Code: 50% (of which tests are worth 10%)
Report: 50% (of which test description is worth 10%)

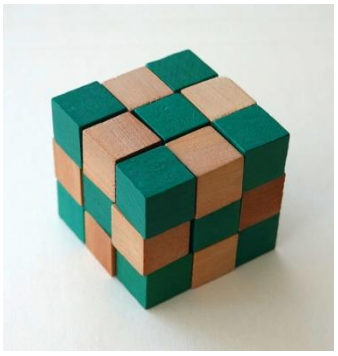
Submit a notice of disruption via ask.mq.edu.au if you are unable to submit on time for medical or other legitimate reasons.

Late penalty without proper justification: 20% of the full marks for the assessment per day or part thereof late.

Overview

The assignment code bundle provides a Scala skeleton program for solving the "snake" puzzle. The snake is a sequence of 27 wooden cubes strung along a piece of string that runs through them.



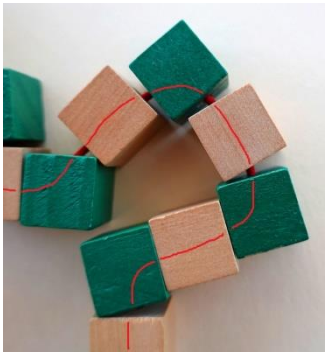


The aim of the puzzle is to arrange the snake so that it forms a 3x3x3 cube.

Other than the two end cubes, each other cube either has the string:

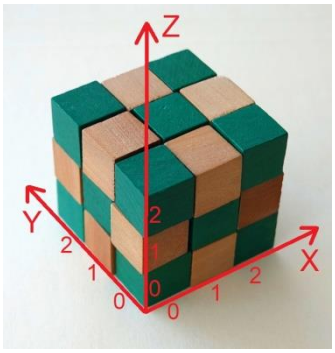
- passing through two opposite faces; or
- passing through two adjacent faces

A cube that has the string passing through two adjacent faces can be rotated into any of four positions (90 degrees apart) that will determine which direction the next segment heads in.



Technical description

We will refer to cube positions using an X-Y-Z coordinate system that will also correspond to indexing a 3D array.

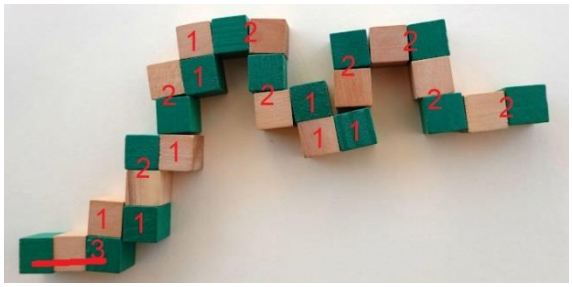


The cube in the bottom corner closest to us is at position (0, 0, 0) and the farthest corner on the top layer is at (2, 2, 2).

We will refer to directions using (as defined in the Scala program):

- XPos – moving along the X-axis in the positive direction
- XNeg – moving along the X-axis in the negative direction
- YPos – moving along the Y-axis in the positive direction
- YNeg – moving along the Y-axis in the negative direction
- ZPos – moving along the Z-axis in the positive direction
- ZNeg – moving along the Z-axis in the negative direction

A snake will be specified by the length of its segments, not including the last cube of the previous segment. In this way, the sum of the segment lengths will always be 27 (= 3 x 3 x 3).



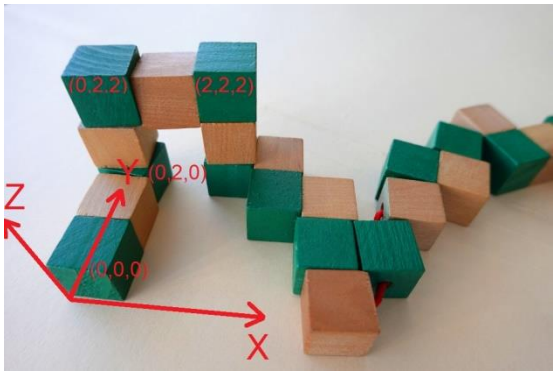
In the picture at right, we can see that the first segment has 3 cubes. The next looks like it might have 2 cubes but we have to exclude the last cube of the first segment; so the second segment has just one cube. Etc.

A solution to the puzzle will specify:

- the position of the first cube (in the 3D coordinate space)
- a list of directions

For example, if we wanted to describe the start of the snake in the picture at right, then we can see that the start position is (0, 0, 0). The first few directions are:

- YPos (going from (0, 0, 0) to (0, 2, 0))
- ZPos (going from (0, 2, 0) to (0, 2, 2))
- XPos (going from (0, 2, 2) to (2, 2, 2))
- ZNeg (going from (2, 2, 2) to (2, 2, 0))
- (then it wanders out of the 3x3x3 cube)



Note that the snake isn't required to start from (0, 0, 0).

Your task will be to find all solutions for a given snake. To prevent symmetrical solutions, we have the following restrictions:

- the start position (x, y, z) must have y < 2 and z = 0
- the first direction must be XPos (so the picture above has an invalid first direction)
- the second direction must be YPos

What you have to do

The skeleton code bundle contains:

- Main.scala (in src/main/scala); no changes required to this (although you may what to do some informal testing from the main function
- Snake.scala (in src/main/scala); flesh out the skeleton functions here
- SnakeTests.scala (in src/test/scala); add your test cases here (in this file, not a separate file)

The Scala program

The skeleton code bundle includes the functions and data structures you will need to solve the puzzle (in Snake.scala).

The directions are defined with the names as above.

A snake is specified as a list of integers. The one corresponding to the picture above with segment numbers is:

```
val snake = List(3, 1, 1, 2, 1, 2, 1, 1, 2, 2, 1, 1, 1, 2, 2, 2, 2)
```

The state of the cube is represented by the CubeState class:

```
  sealed abstract class CubeState
  case class StartState(segments: List[Int], start: (Int,Int,Int))
                                                        extends CubeState
  case class OtherState(segments: List[Int], occupied:Set[(Int,Int,Int)],
                        end: (Int,Int,Int), dir: Direction, numMoves: Int)
                                                        extends CubeState
```

The StartState is used before anything has been put into the 3x3x3 cube. It contains:

- the list of segment lengths in the snake
- the start position in the cube for the snake

Note that an X-Y-Z position can be directly represented in Scala as a 3-tuple (x, y, z) which has type (Int,Int,Int).

The OtherState is used for any cube state after the first move (i.e. positioning of the first snake segment). It contains:

- the segments of the snake (all of them, not just the ones still to be used)
- what parts of the cube are occupied
- the end position of the last move so far
- the direction of that last move
- the number of moves so far (i.e. how many snake segments have been placed)

Each “move” corresponds to the placing of each snake segment in the cube. (Segments are placed in the cube in the order they appear in the snake.)

The occupation of the cube could have been represented by a three-dimensional array of Boolean values (true = occupied, false = unoccupied). Instead we have a set of positions (3-tuples). If a position is in the set then that position in the cube is occupied; otherwise not occupied.

The top-level function for finding all the solutions is:

```
def solveSnake(segments: List[Int]): List[((Int,Int,Int), List[Direction])]
```

It takes a snake segment list and returns a list of solutions. A single solution has the form:

```
((Int,Int,Int), List[Direction])
```

It is a 2-tuple where the first part is the start position of the snake and the second part is the list of directions for the segments.

This function will try all possible first moves (subject to the symmetry restrictions) and it will pursue the valid ones.

The function tryMove is used to determine whether a specific move (a direction) can be applied to the current state of the cube:

```
def tryMove(state: CubeState, move: Direction): Option[CubeState]
```

As can be seen, the function either returns the next cube state, indicating the move was valid, or it fails.

The solveSnake function relies on the function getSolutions to recursively look for solutions:

```
def getSolutions(state: CubeState): Option[List[List[Direction]]]
```

This function will see what directions are possible for the current state and try them (tryMove) and recurse . It returns either returns failure or a list of solutions from the current state.

There are some useful helper functions …

```
def getPossibleDirections(state: CubeState): List[Direction]
```

This produces a list of possible directions that can be used in the current state. Note that this function must:

- rely on only the state's direction and numMoves fields
- order the directions in the order they are listed in the definition of class Direction (to allow for predictable marking)

```
def prependDir(dir: Direction, solutions: List[List[Direction]]): List[List[Di
rection]]
```

This takes a list of lists of directions and puts the specified direction at the font of each of the lists.

```
def prependStart(xyz: (Int,Int,Int), solutions: List[List[Direction]]):
                                              List[((Int,Int,Int), List[Direction]
)]
```

This takes a list of lists of directions and turns each of the lists into a 2-tuple where:

- the first part is the specified position (xyz)
- the second part is the original list

```
def getIncrement(dir: Direction):(Int,Int,Int)
```

This function takes a direction and produces a 3-tuple that says how to increment x, y and z when moving in that direction.

Functional programming

While the aim of the programming is to solve the puzzle, you will also be assessed on how well your program is written in the style of functional programming. For example:

- use val instead of var for variable declarations
- avoid iterative loops (for/while); use map, filter, reduce, …
- if you have to use a FOR loop, consider using: for(…) yield …
- use match instead of multiple IFs

Running the program

The skeleton for this assignment is designed to be run from within sbt. In the command prompt for sbt, there are three relevant commands:

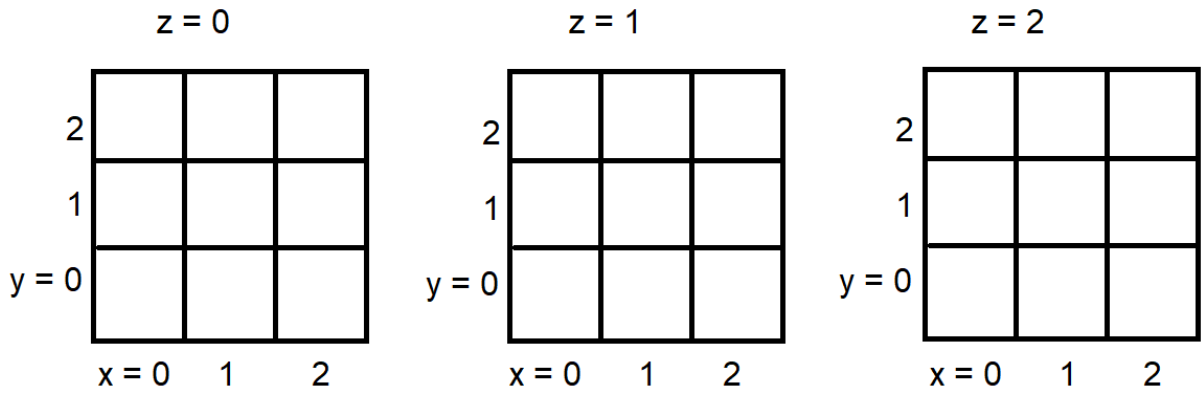
- test – runs the test suite (tests are in src/test/scala/SnakeTests.scala
- run – executes the main function in src/main/scala/Main.scala
- compile – compiles any changed source files; there is no need to separately compile because the run and test command always do any required compilation

Testing

Testing accounts for 20% of the assignment's marks, 10% for the test cases you write and 10% for your report of the testing (mainly focusing on why you have added your test cases).

Your test cases should be added at the end of src/test/scala/SnakeTests.scala after the FIXME comment.

You will see that there are functions for inputting and displaying the cube: see show and toSet. These rely on visualising the cube as three layers (z=0, z=1, z=2):



and using 1=occupied and 0=not occupied.

What you must hand in and how

1. A zip file containing all of the code for your project and a type-written report.

   Submit every source and build file that is needed to build your program from source, including files in the skeleton that you have not changed. Do not add any new files or include multiple versions of your files. Do not include any libraries or generated files (run the sbt "clean" command before you zip your project). We will compile all of the files that you submit using sbt, so you should avoid any other build mechanisms.

2. Your submission should include all of the tests that you have used to make sure that your program is working correctly. Note that just testing one or two simple cases is not enough for many marks. You should test as comprehensively as you can.

3. Your report should describe how you have achieved the goals of the assignment. Do not neglect the report since it is worth 50% of the marks for the assignment.

Your report should contain the following sections:

- A title page or heading that gives the assignment details, your name and student number.
- A brief introduction that summarises the aim of the assignment and the structure of the rest of the report.
- A description of the design and implementation work that you have done to achieve the goals of the assignment. Listing some code fragments may be useful to illustrate your description, but don't give a long listing. Leaving out obvious stuff is OK, as long as what you have done is clear. A good rule of thumb is to include enough detail to allow a fellow student to understand it if they are at the stage you were at when you started work on the assignment.
- A description of the testing that you carried out. You should demonstrate that you have used a properly representative set of test cases to be confident that you have covered all the bases. Include details of the tests that you used and the rationale behind why they were chosen. Do not just print the tests out without explanation.

Submit your code and report electronically as a single zip file called ass1.zip using the appropriate submission link on the COMP3000 iLearn website by the due date and time. Your report should be in PDF format and should sit in the top directory of project (where build.sbt sits).

DO NOT SUBMIT YOUR ASSIGNMENT OR DOCUMENTATION IN ANY OTHER FORMAT THAN ZIP and PDF, RESPECTIVELY. Use of any other format slows down the marking and may result in a mark deduction.

Marking

The assignment will be assessed according to the assessment standards for the unit learning outcomes.

Marks will be allocated equally to the code and to the report. Your code will be assessed for correctness and quality with respect to the assignment description. Marking of the report will assess the clarity and accuracy of your description and the adequacy of your testing. 20% of the marks for the assignment will be allocated to testing.