# Loops & reducing problems to smaller problems

## Dirty Numbers:

A dirty number is a positive integer that is only divisible by 1, 2, 3, or 5. (2,3,5 are its own prime factors).

Write a method that can determine if a number is a dirty number. Input: number, output: Boolean value

Hints:

* Are there cases you can immediately solve?
* For the cases you cannot immediately solve; is there a way to reduce them to a case where you do know the answer?
  + Is there a way to write a loop to continuously reduce the problem to a smaller problem?
* The modulo operator may be useful, in Java and Python modulo is represented by “%”, in R they use “%%”:
  + Modulo returns the reminder of a long division, this can be used to quickly check if a number is divisible by another number.
  + Some examples:
    - 12 % 5 = 2
    - 14 % 5 = 4
    - 15 % 5 = 0

Bonus questions:

* Rewrite your answer to a recursive loop
* Can you optimize your solution to loop as few times as possible?

Roman numerals:  
Convert roman numerals to Arabic and back.

Write two methods, one to convert from roman to Arabic, and one to convert back.

Roman numerals will use a String value, Arabic will use an integer value.  
  
Roman numbers:

I = 1

V = 5

X = 10

L = 50

C = 100

D = 500

M = 1000

IV = 4

IX = 9

XL = 40

XLIX = 49  
XC = 90

CDL = 450

MCMLXXVI = 1976

Etc.  
  
Hints:

* Similar to the previous problem;
  + are there simple cases where you can easily determine the final solution?
  + When looking at bigger numbers are there ways to reduce the problem to a smaller number until you reach a stop-scenario?
  + Can this be made into a loop?
  + Can you solve both directions with the same loop (e.g. both need a while loop) or is it better/necessary to use different loops?
  + Pay special attention to cases where 2 roman numerals interact (e.g. IV, IX) to create 1 Arabic number.

## Trees 1)

Create a tree in an OO manner using a Node class.

Questions:

* How to determine if a node is a leaf node or not.
* What is your root node?

## Trees 2)

Find the maximum depth of a given tree.

* How do you efficiently traverse a tree?
* Can you use recursion?
* Experiment with uneven trees.

## Trees 3)

Each leaf node contains a number value. Find the biggest value in your tree.  
Can you re-use the tree traversal from the depth exercise?

## Trees 4)

What is the shallowest depth at which a certain number can be found in the tree?  
Return 0 if the number is not present.

* What happens if the same number is present in multiple leaf nodes?

## Trees 5 & inheritence)

Instead of numbers the leaf nodes now contain animals.  
Create your animal objects in an OO manner using a super class “animal” and at least 2 animal subclasses e.g. “Cat” and “Dog”.

The class “Animal” contains a method “makeANoise” which makes the noise of the specific animal. Give the parent clash a generic noise.  
Overwrite this method in your animal subclasses using “inheritance” with the noise of that specific animal (e.g. the dog class goes “Woof”)

Now perform the following exercises:

* Print the noise the animal in a leaf node X makes.
* Find the shallowest depth that contains a dog.

Make it generic

## Trees 6:

Create a tree of animals, now print the animal sounds from left to right.  
Write this in a generic way.

## Matrices

* Create a matrix
* Define a start and end point
* Define certain cells as blocked
* Measure the distance between start & end using the Manhattan distance, blocked cells are impassable.

## Matrices 1:

Write generic code to be able to block arbitrary cells in an efficient way.  
Make sure this is written in such a way that you can draw any combination of blocked cells.

Try to make this approach efficient. Simply creating a method that blocks a specific coordinate is fine for a 5x5 matrix. But not for a 1000x1000 matrix. Create it in such a way that entire sections can be marked as blocked at once. Consider the (dis)advantages different object designs for blocking entire sections; 1 object that marks all “blocked” cells, an object per “blocked” subshape, an object per line etc.

Try multiple different shapes. At minimum try to create the following shapes of blocked cells: L, C, O, +. Also rotate and invert these shapes.

## Matrices 2:

Find the path between the start and end. Only focus on paths that are reachable.

Begin with a start and end that are directly next to eachother, then increase the distance.  
Additionally begin with solving configurations where the start is to the left of the end.  
Then try configurations where the start is to the left, and above, the end.  
Then try configurations where the start is to the right & below the end.  
  
Additionally, first solve it without obstructions before adding “blocked” cells that require a detour.  
  
When incorporating blocked cells also consider different configurations. For example, consider the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  | Start |  | end |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | end |
|  |  |  |  |  |
| Start |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Start |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  | end |

Also consider different size matrices and ensure your pathfinding approach still works on larger matrices (up to 100x100)

Extra challenge: use maps that are not rectangular in shape.

## Matrices 3:

What happens when the path is blocked?

## Matrices 4 (advanced):

Guarantee that you find the shortest path (if a path exists).

## Three in a row:

Program a simple three in a row game. Follow the following substeps:

## Three in a row: the base game

Use a matrix to represent the board. Include a generic method a player can use to make a move. Now create a permanent loop that alternates between the players for input using the command line interface using the board coordinates. At the end of each move, check if the game is finished before asking the next player to make a move.

## Three in a row: adding an AI

Create a generic representation of an AI player using OO programing. When it is player 2’s turn, instead of asking for a move via the command line, simply have the AI make a move. Ensure the AI always plays a legal move, but the move does not need to be good.

## Three in a row: implement a clever AI (advanced).

Implement a min-max search for the AI’s move ( <https://en.wikipedia.org/wiki/Minimax> contains the pseudocode). This requires recursive methods.  
This can be optimized using alpha-beta search for an additional challenge.

## Three in a row: no, N in a row.

Make it possible to play at different board sizes.