Homework assignment #1

Please submit a `.zip` archive containing a single file `homework1.hs` with your solutions to the problems below. Moodle blocks file submissions with `.hs` extension.

Your code will be evaluated based on correctness, clarity, conciseness, and adherence to the functional programming paradigm. For this homework assignment, only `import Prelude`.

Good luck! 🍀

Grading will be partially automated. Please ensure that you submit exactly one file named *homework1.zip*, containing a single file named *homework1.hs*. Do not alter the function names provided, and strictly adhere to the specified type signatures. Any deviations will result in automated testing failures and impact your grade.

Problem A: Magical Potion Mixing 🧪

In a magical alchemy lab, you have two cauldrons, each containing a sequence of magical ingredients. Your goal is to create a new potion by interleaving ingredients from both cauldrons. However, there's a twist: you must mix the ingredients in a specific pattern to produce the final potion. The potion should be created by alternating ingredients from the two cauldrons. If one cauldron runs out of ingredients, you simply continue with the remaining ingredients from the other cauldron.

Write a Haskell function `mixPotions :: [a] -> [a] -> [a]` that takes two lists of ingredients, representing the contents of the two cauldrons, and returns a new list with ingredients for the new potion.

```
Input: ["Unicorn Horn", "Dragon Scale", "Phoenix Feather"] ["Mermaid's
Tear", "Goblin Ear"]
Output: ["Unicorn Horn", "Mermaid's Tear", "Dragon Scale", "Goblin Ear",
"Phoenix Feather"]

Input: ["Pixie Dust"] ["Fairy Wing", "Elf Ear", "Goblin Tooth"]
Output: ["Pixie Dust", "Fairy Wing", "Elf Ear", "Goblin Tooth"]

mixPotions :: [a] -> [a] -> [a]
mixPotions = undefined

testA1 = mixPotions ["Unicorn Horn", "Dragon Scale", "Phoenix Feather"]
["Mermaid's Tear", "Goblin Ear"] == ["Unicorn Horn", "Mermaid's Tear",
"Dragon Scale", "Goblin Ear", "Phoenix Feather"]
testA2 = mixPotions ["Pixie Dust"] ["Fairy Wing", "Elf Ear", "Goblin Tooth"]
== ["Pixie Dust", "Fairy Wing", "Elf Ear", "Goblin Tooth"]
```

Problem B: Digital Root #️⃣

The digital root of a number is the value obtained by an iterative process of summing digits until a single digit is achieved. For example, the digital root of 4935 is calculated as follows:

```
4 + 9 + 3 + 5 = 21
2 + 1 = 3
So, the digital root of 4935 is 3.
```

(1) Create a Haskell function `digitalRoot :: Int -> Int` that takes a positive integer and returns its digital root.

```
Input: 10
Output: 1

Input: 12345
Output: 6

testB1 = digitalRoot 3 == 3
testB2 = digitalRoot 1408 == 4
```

(2) Write a Haskell function `digitalRootsList :: [Int] -> [Int]` that takes a list of positive integers and returns a list of their digital roots.

```
Input: [4935, 12345, 987]
Output: [3, 6, 6]

testB3 = digitalRootsList [2, 23, 654, 12, 39, 12347] == [2, 5, 6, 3, 3, 8]
```

Problem C: Tower of Hanoi 💿

The Tower of Hanoi is a classic mathematical puzzle featuring **three rods** and `n` different-sized disks. The challenge is to move a complete stack of disks from one rod to another while adhering to the rule that larger disks can never be placed on top of smaller ones.

Write a function `hanoi :: Int -> Int` that computes the number of steps needed to move the whole stack of `n` discs from one rod to another using recursion.

Place a comment in the code explaning your strategy!

```
{-
Strategy: ??
-}

hanoi :: Int -> Int
hanoi = undefined

testC1 = hanoi 3 == 7
testC2 = hanoi 14 == 16383
```

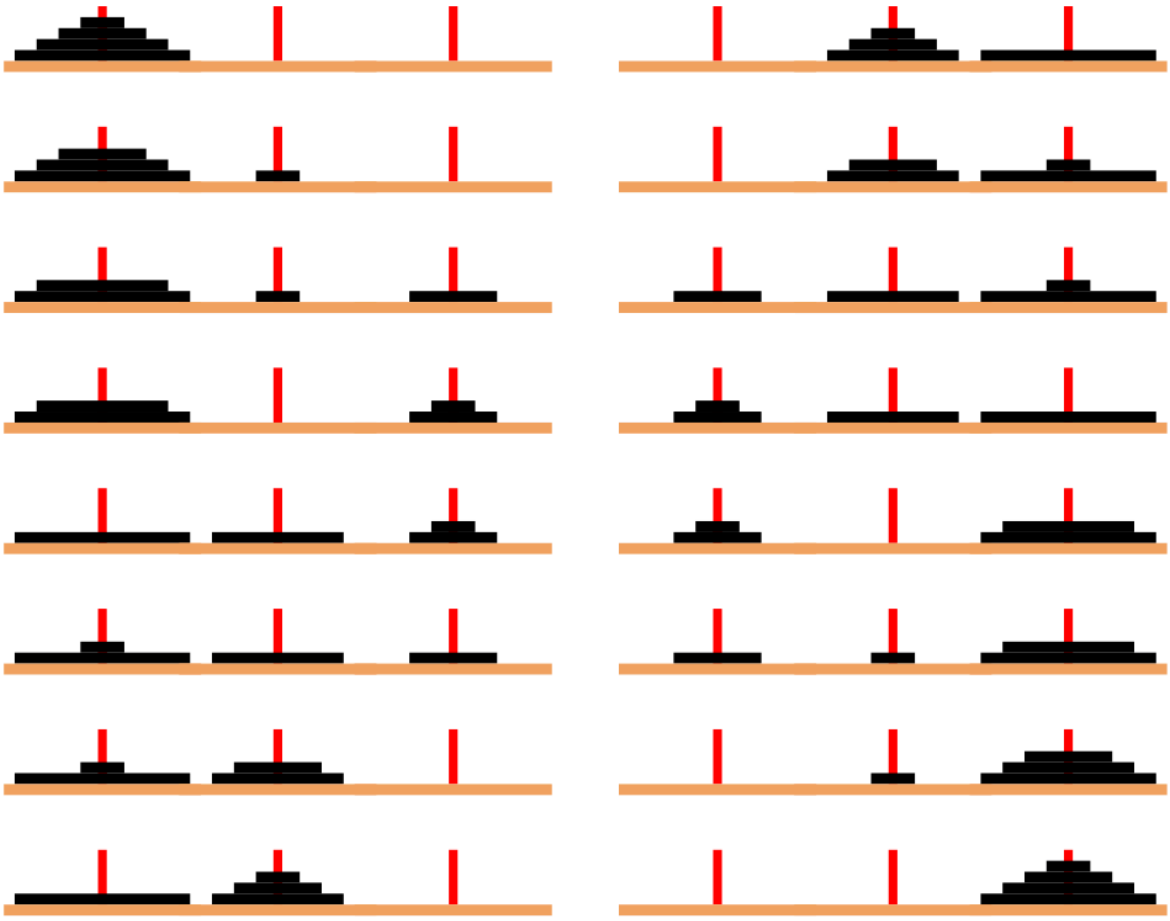


*Image credit: Wolfram MathWorld

Problem D: Music Theory 🎼🎹

Music theory is regulated by multiple rules and concepts to build harmonious melodies.

One of such concept is the **scale**. There are many types of scales such as the **major** scale or the **pentatonic** scale.

The goal of this problem is to generate a string representing a scale (conventional or not) described by a given pattern and a root note.

However, before continuing, more musical knowledge is required to understand the problem better.

Piano keys

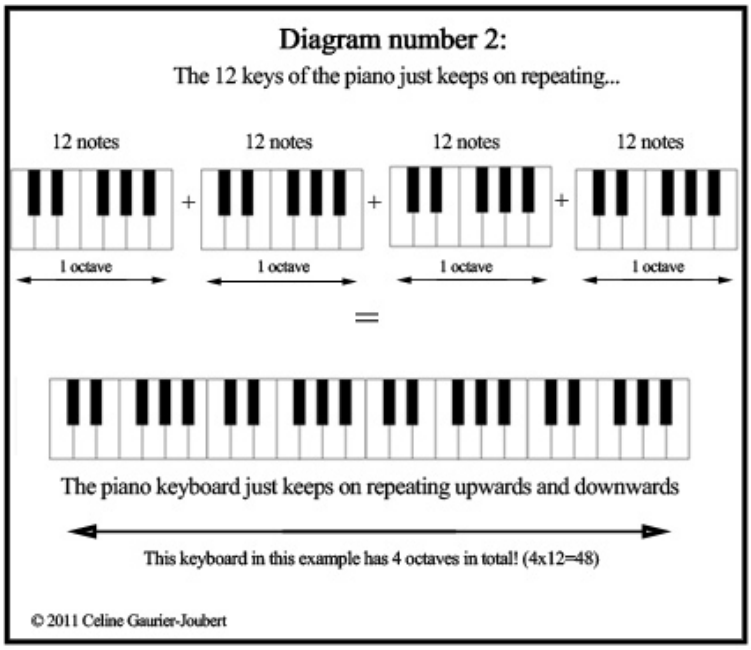
A piano key is one of the levers on a piano that you press to produce sound. There are two types of keys on a piano:

White Keys: These are the larger keys (C, D, E, F, G, A, B).

Black Keys: These are the smaller keys representing sharp (#) and flat (♭) notes.

As there is an equivalence between flat notes and sharp notes, we will only use the sharp notation.

For example, the black key to the right of C is C# (C sharp) which is equivalent to D♭.



A piano is typically composed of 12 ordered keys (C, C#, D, D#, E, F, F#, G, G#, A, A# and B) that are repeated (octaves).

The first part of this problem consists in generating `infiniteKeys :: [String]`, an **infinite** list of strings representing the 12 ordered keys in a piano:

```
Input : Nothing

Output :
["C","C#","D","D#","E","F","F#","G","G#","A","A#","B","C","C#","D","D#","E",...]
```

Scales

A scale is a sequence of musical notes arranged following a specific pattern.

For example, the **major scale** follows the pattern : **W-W-H-W-W-W-H**

where W represent a **whole** step (or 2 keys), and H is a **half** step (or 1 key).

The goal of this part is for you to write a recursive function `musicScale :: String -> String -> String` that takes as input a root note (like C), the pattern, and outputs the notes that belong to the scale defined by that pattern.

Example Calculate the G major scale:

```
Input : "G" and "WWHWWWH"

Output : "G-A-B-C-D-E-F#-G"
```

- We start with the note G
- We count a whole step (2 keys : G# then A), and get A
- From A, we count another whole step (A# then B), and get B.
- From B, since the 12 keys are repeated, we count a half step (a single key : C), and get C, and so on.

Finally, we get "G-A-B-C-D-E-F#-G"

Consider that patterns can only contain whole steps and half steps and are of arbitrary length. If the root note or pattern is invalid, your program should return an error. Any pattern made up solely of whole and half steps is considered valid.

```
testD1 = musicScale "G" "WWHWWWH" == "G-A-B-C-D-E-F#-G"
```