Homework assignment #2

Please submit a `.zip` archive containing a single file `homework2.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 *homework2.zip*, containing a single file named *homework2.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.

Acknowledgement: The problems of this homework assignment were suggested by Antonia CUBA & Maeva CHEKAM.

Problem A: Project Team Formations 💼

In this exercise, you will create functions to form project teams from a list of candidates, ensuring that all team members possess a specific skill. The exercise consists of several steps that build on each other.

[Part 1] Combinations

Implement function `combinations` that generates all possible combinations of a given length from a list of items (candidates).

In [ ]:
```haskell
combinations :: Int -> [a] -> [[a]]
combinations = undefined

testA1 :: Bool
testA1 = combinations 2 ["Alice", "Bob", "Charlie"] == [["Alice", "Bob"]
                                                       ,["Alice", "Charlie"]
                                                       ,["Bob", "Charlie"]]
```

[Part 2] The necessary skills 👩‍🚒

Define the type `Candidate` (using a `type` alias) compatible with definitions `candidates1` and `candidates2` below, and write a function `allHaveSkill` that checks whether all members of a team (a list of candidates) have the selected skill.

`allHaveSkill :: String -> [Candidate] -> Bool`

In [ ]:
```haskell
type Candidate = undefined

allHaveSkill :: String -> [Candidate] -> Bool
allHaveSkill = undefined

candidates1 :: [Candidate]
candidates1 = [("Alice", ["Python", "Java"])
              ,("Charlie", ["Python", "C++"])]

candidates2 :: [Candidate]
candidates2 = [("Alice", ["Python", "Java"])
              ,("Bob", ["JavaScript"])]

testB1 :: Bool
```

```
testB1 = allHaveSkill "Python" candidates1

testB2 :: Bool
testB2 = not (allHaveSkill "JavaScript" candidates2)
```

[Part 3] Specialized teams

Write a function `validTeamCombinations` that takes a list of candidates and a required skill, and outputs all possible teams where every team member possesses the specified skill. Reuse code from Part 1 and 2 as much as possible.

Only the names of the candidates should be included in the teams.

HINT: 💡 Higher-order functions like `map`, `filter` and `concatMap` might prove useful here.

In [ ]:
```
validTeamCombinations :: [Candidate] -> String -> [[String]]
validTeamCombinations = undefined


candidates3 :: [Candidate]
candidates3 = [("Alice", ["Python", "Java"])
              ,("Bob", ["JavaScript"])
              ,("Charlie", ["Python", "C++"])]

candidates4 :: [Candidate]
candidates4 = [("Alice", ["Python"])
              ,("David", ["Python", "JavaScript"])]

testC1 :: Bool
testC1 = validTeamCombinations candidates3 "Python" == [["Alice"]
                                                       ,["Charlie"]
                                                       ,["Alice","Charlie"]]

testC2 :: Bool
testC2 = validTeamCombinations candidates4 "Python" == [["Alice"]
                                                       ,["David"]
                                                       ,["Alice","David"]]
```

[Part 4] Counting teams 1️⃣2️⃣3️⃣

Implement the `countValidTeams` function that counts how many different teams can be formed where *all* members have the required skill.

In [ ]:
```
countValidTeams :: [Candidate] -> String -> Int
countValidTeams = undefined

testD1 :: Bool
testD1 = countValidTeams candidates3 "Python" == 3

testD2 :: Bool
testD2 = countValidTeams candidates4 "JavaScript" == 1
```

Problem B: Strawberry harvest 🍓

A farmer wants to harvest ripe strawberries in a square garden of size $n \times n$.

Only k plants in the garden have ripe strawberries, and the farmer aims to harvest them all.

The garden is represented as a 2D list (a list of lists), where each *element* corresponds to a location in the garden. It means that the list at index i represents the i-th row of the garden, and the *element* at index j in that row represents the location at row i, column j of the garden (see image below).

- A location with ripe strawberries (which the farmer must visit) is represented by 1.
- A location without ripe strawberries (which the farmer may skip) is represented by 0.

The following rules define how the farmer can move in the garden:

- The farmer must start at the top-left corner of the garden (row 0, column 0) and finish at the bottom-right corner (row $n-1$, column $n-1$) where his car is parked. He moves through the garden in one direction (left to right, following a row). He moves left to right along each row and can advance either by taking a single step forward or by jumping three steps at a time, skipping two locations. Thus, he can move either by **1** step or by **3** steps forward at a time.
- A location is considered visited if the farmer moves directly to it from his previous position, using either move. Locations skipped by jumping are **not** considered visited.
- When the farmer moves over the end of a row, he automatically appears at the start of the next row. This transition counts as a single step, and can be taken by any of the moves: Making a single step from the last position moves the farmer to the first position in the next row; making a jump from the one-before-last position moves the farmer to the second position of the next row, etc.

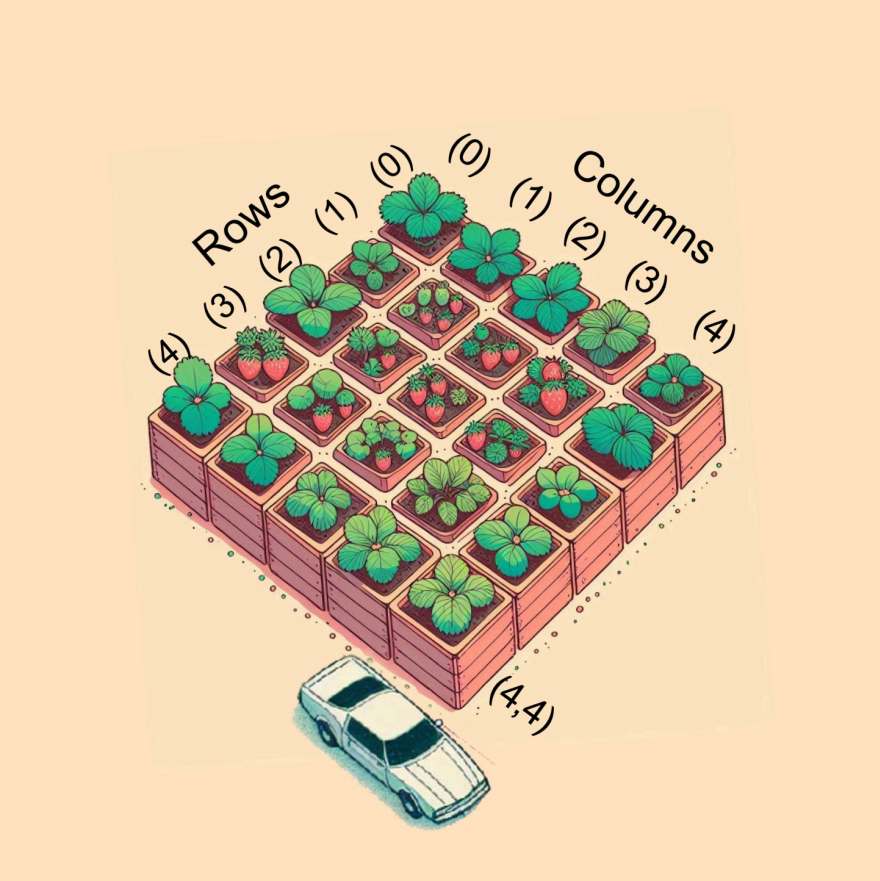
The goal of this problem is to determine how many different sequences of moves the farmer can use to traverse the whole garden, ensuring that all locations with ripe strawberries are visited.

Example

The garden shown in the picture below can be represented as :

```
garden = [[0,0,0,0,0],[0,1,1,1,0],[0,1,1,1,0],[1,1,1,0,0],[0,0,0,0,0]]
```

Here, k = 9



[Part 1] Extract relevant locations

Use **foldr** to write a function `relevantLocations` that outputs the list of locations that **must** be visited by the farmer. These must-be-visited locations include the starting location, the ending location, and locations of all ripe strawberries.

Here, each location should be represented by a single integer, instead of using a pair of integers (row and column). This new integer index corresponds to the position that the location would have if all rows of the 2D garden were connected into one long row.

Example

Considering the previous garden:

```
relevantLocations garden  == [0,6,7,8,11,12,13,15,16,17,24]
```

Note here that the first element (0) is the index of starting location, and the last element (24) is the index of the ending location.

In [3]:
```haskell
relevantLocations :: [[Int]] -> [Int]
relevantLocations = undefined
```

[Part 2] Gardenacci

Write a function that calculates the n-th Gardenacci number.

The Gardenacci sequence is given by:

- $g(0) = 1$
- $g(1) = 1$
- $g(2) = 1$
- $g(n) = g(n-1) + g(n-3)$

(Striking similarity to the Fibonacci sequence is not accidental.)

In [ ]:
```haskell
gardenacci :: Int -> Integer
gardenacci = undefined
```

[Part 3] Possible traversals

In this part, you need to write a function `numberPaths` to calculate how many paths the farmer can take to traverse the garden, ensuring that all the relevant locations (where ripe strawberries are) are visited.

Input:

- The dimension n of the garden (n is the number of rows or columns in the garden).
- The garden, provided as a `[[Int]]`. Locations with ripe strawberries are marked with `1`.

Output:

- The number of possible ways the farmer can move from the first location to the last location, while visiting **all** strawberry plants exactly once, and following the movement constraints (1 or 3 steps forward).

HINT: 💡 The number of possible paths of length m is related to the m-th Gardenacci number. Like the Fibonacci sequence, this sequence also grows exponentially. So, even for a relatively small garden, the count may exceed the limits of the `Int` type. It might be useful to recall how to efficiently compute the Fibonacci sequence from class.

In [ ]:
```haskell
numberPaths :: Int -> [[Int]] -> Integer
numberPaths = undefined

testE1 :: Bool
testE1 = numberPaths 2 [[0,0],[0,1]] == 2

testE2 :: Bool
```

```haskell
testE2 = numberPaths 3 [[0,0,0],[1,0,0],[1,0,0]] == 4

testE3 :: Bool
testE3 = numberPaths 5 [[0,0,0,0,0],[0,0,0,0,0],[0,1,0,0,0],[0,0,0,0,0],[0,0,1,1,0]] == 1681
```