-

1. Please submit exactly one Haskell file where the questions are separately by a comment line. Add the answers to additional questions as well as your Haskell outputs where requested as comments.
2. Please put your group number, names and student numbers at the beginning of file.
3. Marks will be awarded for correct functionality, for good code quality, for complying with the instructions above and if asked for presenting your work.
4. complying with the instructions includes: Submitting exactly 1 Haskell files that compliles
5. readable presentable layout of the file. Code quality Having added comments where requested.
6.

Question 1. a) Write a Haskell function `average` that computes the average of three numbers. (Note: For simplicity you may use input type `Float`.) Add a function called `howManyAboveAverage1` that computes how many of these three numbers are above the average. Give a second solution (`howManyAboveAverage2`) for this function which is different in nature, i.e., uses an algorithm that is computationally different. Run your programs with your student numbers, and include the Haskell call and the original output as a comment into your solution.

b) The following demonstrates a method how you could check the correctness of your programs: Add the following code to the top of your file `import Test.QuickCheck` and write a function as below (please complete its type) that compares the two implementations in a) of your howManyAboveAverage functions:

```
checkcorrectness x y z =
     howManyAboveAverage1 x y z  == howManyAboveAverage2 x y z
```

In the commandline then do the check by typing.

```
quickCheck checkcorrectness
```

Include the response of Haskell as a comment.

Question 2. Pizzeria Alfredo sells pizzas of an arbitrary size and with an arbitrary number of toppings. The owner Alfredo wishes to have a program that allows to compute the selling price of a pizza depending on its size (given by its diameter in cm) and the number of toppings. The pizza base costs £0.001 per cm^2 and the costs for each topping are £0.002 per cm^2. Since Alfredo also wants to make some profit, he multiplies the costs of a pizza by a factor 1.5. Can you help Alfredo with a suitable Haskell function?
Finally, add a second function concerned with the answer of the following: Pizza Bambini (tomatoes, mozzarella, ham, salami, broccoli, mushrooms, 16 cm) is more expensive than Pizza Famiglia (tomatoes, mozzarella, 32 cm)? Add your response as a comment, together with a suitable Haskell output that justifies your response.

Question 3. Starting from the programs

```
divides :: Integer -> Integer -> Bool
divides x y = y `mod` x == 0

prime :: Integer -> Bool
prime n = n > 1 && and [not(divides x n) | x <- [2..(n-1)]]

allprimes :: [Integer]
allprimes = [x | x<- [2..], prime x]
```


Define a function `allprimesBetween:: Integer -> Integer -> [Integer]` that produces all prime numbers between two bounds.
Next produce an infinite list `primeTest:: [Bool]` such that the nth position of the list is `True` if n is prime, and `False` otherwise. Then, produce an infinite list such that the nth position is the pair (n, b) where the value b is `True` if n is prime and `False` otherwise.

Finally, add a function `primeTwins` that, given n, computes how many prime twin pairs are amongst the first n prime numbers. Use this to answer the question: How many prime twin pairs are there amongst the first 2000 prime numbers? [Hint: a prime twin pair consists of two prime numbers that have a difference of 2. Examples: (5,7), (11,13)]

Question 4. In cryptography, we often need exponentiation modulo a number n. Write a (recursive) program `expmod` that takes integers M, e, n as input and computes

$$M^e \bmod n.$$

Note: the computation of the huge number M^e can be avoided using the fact that the modulo operation commutes with multiplication, in particular:

$$M^{(e+1)} \bmod n = ((M^e \bmod n) * M) \bmod n.$$

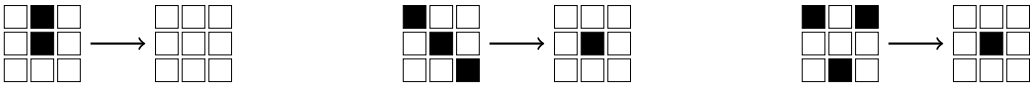
Implement an improved second version `expmodfast` using the observation that $M^{2*e} = (M^e)^2$? Demonstrate your function with your student number as exponent. (Result as comment!)

Question 5 ***("Petri dish").*** Implement a simulation of cells on a Petri dish as follows. The dish is represented as a rectangular grid where each point is a cell. Each cell is either alive or dead and has up to 8 neighbours:

The simulation evolves in steps and the fate of each cell is decided according to the following rules:

1. A *live* cell, which
   (a) has fewer than 2 live neighbours *dies* because of underpopulation,
   (b) has 2 or 3 live neighbours *lives*,
   (c) has more than 3 live nieghbours *dies* because of overpopulation.
2. If a cell is *dead* and has exactly 3 live neighbours, it becomes a *live* cell.

Examples (black = live, white = dead):



You should represent the grid as a *list of lists* of `Bool`, that is a list that contains rows of cells. For the simulation implement the following:

a) a function `valid :: [[Bool]] -> Bool` which checks whether the given grid is valid. A grid is valid if and only if it has a rectangular shape (lengths of all rows are the same), and if it is at least 3×3.

b) a function `showGrid :: [[Bool]] -> String` that returns a string representation of a grid and a function `printGrid :: [[Bool]] -> IO ()` that prints the grid to the terminal. You can use `O` to represent a live cell and `.` to represent a dead one. Example output:


```
.O...O.
..O.O..
...O...
..O.O..
.O...O.
```

c) a function `readGrid :: String -> [[Bool]]` that reads a string representation of a grid and returns the corresponding grid.

d) a collection of functions that implement the evolution of one grid state into another. In particular, you need to implement the following functions:

- `neighbours :: [Bool] -> Int` which takes a list of neighbours of a cell and returns the number of live neighbours,
- `newCell :: Int -> Bool -> Bool` which takes the number of neighbours of a cell and the cell itself and decides if the cell is to live or die after the step, and
- `step :: [[Bool]] -> [[Bool]]` which takes a grid and returns a new grid which is a one step evolution of the old one. This function should make appropriate use of above two functions and we encourage you to break it down into multiple smaller functions that you can test in isolation.

There are at least 2 ways you can approach walking the grid. As a hint, for both approaches you will need to think about how to deal with boundary areas of the grid. The step function has to validate the grid before processing it. If the grid is invalid, it should return an error by calling `error "Invalid grid"`.

e) a function `loadGridFile :: String -> IO [[Bool]]` which reads a grid from the given file. You can assume that the file contents were generated using `showGrid`. Validate the grid after loading it and signal an error if it is invalid.

f) a function `interactive :: IO ()` which asks the user for a file name, loads a grid from the given file, prints it, and waits for user input. If the user presses ⟨*Enter*⟩, the function prints the next step in the evolution and waits for input again. If the user writes `exit` (and presses ⟨*Enter*⟩), then the function terminates.

g) improvements to the simulation. After you have completed the main simulation, you may improve it. Here are some ideas - choose one of them - they are of increasing difficulty (and marks). They should be implemented as new functions – for example, if you want to improve on `interactive`, name the improved function `interactive1`, etc. Always include a comment about what the improvement is and how you approached the implementation, otherwise your improvement may be overlooked.

- In `interactive`, allow the user to save the current grid in a file that the user chooses.
- In `interactive`, give the user the option to generate a random grid of a given size, instead of loading it from a file.
- The file format is quite inefficient and can be improved in a few ways. One way is to specify the grid size and only the significant cells from the top and the left, and then complete the grid after loading. E.g.,

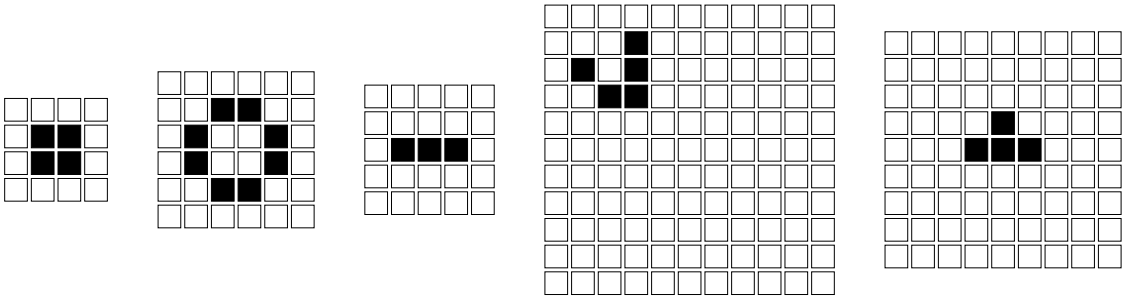
```
.........
.0.0.....
..0.0....
...0.0...
....0....
.........
```

can be represented as the 4 × 8 grid

```
.
.0.0
..0.0
...0.0
....0
```

- Another way of making the grid file format more efficient is based on the observation, that the files contain many repeating characters. This makes them suitable for *run-length encoding*, that is an encoding where repeating sequences of characters are represented as the character and its count. E.g., `HHHHaaaaaaaaaaskell` can be represented as `4H11a1s1k1e2l` or even `4H11aske2l`. Write both an encoding and decoding function for grid representations. You can even identify the grid file format by using different file name extensions.

Finally, here are some patterns that you can use to test your programs. Some of them are stable (i.e., they don't change between steps), some of them oscillate, or move.



You may be asked to present your work in one of the labs following submission.