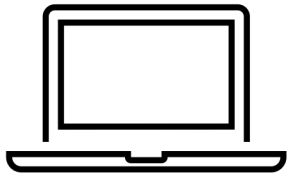
Week 2 exercises

Exercises 1-4 are from the Tuesday session, exercise 5 is from Thursday.

The remaining exercises cover topics from this week and last, and are included for your own practice. If you work ahead and complete them during class, that's great, but you are not expected to do this. You **should** complete them in your own time.

Within these exercises, a reference of the form [Hutton, ex N.X] refers to exercise X from chapter N of Graham Hutton's *"Programming in Haskell, Second Edition"* (2016).

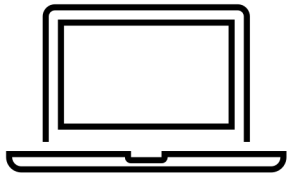
Give it a go!



Now you can encode a single character from the string. (Lecture 5, slides 5&6)

1. How can you encode an entire string?
2. And how do you decode a string?

Give it a go!

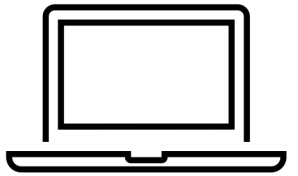


3. Define a function to return the n^{th} number in the Fibonacci sequence

- WHAT IS THE FIBONACCI SEQUENCE??

  1, 1, 2, 3, 5, 8, 13, 21, …

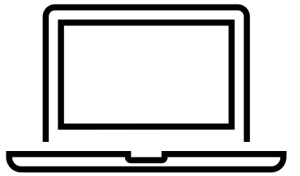
Give it a go!



4. Find the n^{th} item in a list

This duplicates the work of the !! operator
`xs !! n` returns the n^{th} item of xs (starting at the 0^{th})

Give it a go!



The roots of a quadratic equation $ax^2 + bx + c = 0$

are given by $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note that $\sqrt{b^2 - 4ac}$ is used twice (so is $2a$).

5. Write a function **roots**, with type:

```
roots :: (Float, Float, Float) -> (Float, Float)
```

to return the roots of an equation, where the input tuple represents the values of a, b and c in the above equation.

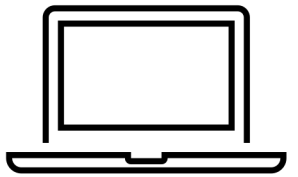
Reminder!

The above four exercises have been set for completion in the lab sessions. You should be practising your functional programming outside class too.

Haskell is available on the managed desktop of all PCs across the university. If you are using a PC in a place not normally used for computer science teaching, use *Software Centre* to install it on that PC.

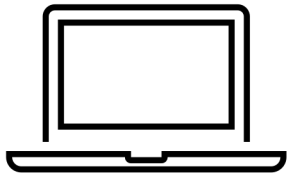
The following exercises should be completed, either in lab classes or your own time. **Don't just look at the solutions when they become available,** understanding how something works is NOT the same as being able to write it yourself.

Give it a go!



6. Define the following functions in a file with their type signatures
   a) stack takes the first element of a list and puts it on the end of a list
   b) range takes a numerical value and checks to see if it is between 0 and 10, returns True if it is False otherwise
   c) addc takes a Char and a String and adds the Char to the beginning of the String
   d) halves takes a list and divides each element in the list by two
   e) capitalizeStart that takes a string as input and returns the same string with the first character capitalized. (If the first character is not a lowercase letter, it should simply return the input string.)
7. Rewrite the list comprehension `[x ^ 2 | x <- [1..20], even x]` using the library functions map and filter.
8. Here is alternative definition for a function that has already been introduced: `(\ (_:xs) -> xs)` What function is it? Make sure that you understand how this lambda expression works.
9. Write a lambda expressions to perform the following tasks:
   a) Increment an integer value
   b) Decrement a value
   c) (Harder!) Check if a value is a prime number

Give it a go!



10. [Hutton, ex. 4.8] The *Luhn algorithm* is used to check bank card numbers for simple errors, such as mistyping a digit, and proceeds as follows:
    1. Consider each digit as a separate number;
    2. Moving left, double every other number from the second last;
    3. Subtract 9 from each number that is now greater than 9;
    4. Add all the resulting numbers together;
    5. If the total is divisible by 10, the card number is valid.

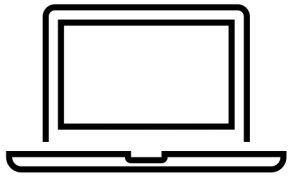
    Define a function `luhnDouble :: Int -> Int` that doubles a digit and subtracts 9 if the result is greater than 9. For example:

```
> luhnDouble 3
6
> luhnDouble 6
3
```

    Using luhnDouble and the integer remainder function mod, define a function `luhn :: Int -> Int -> Int -> Int -> Bool` that decides if a four-digit bank card number is valid. For example:

```
> luhn 1 7 8 4
True
> luhn 4 7 8 3
False
```

Give it a go!



11. In this week's lectures, we examined the following implementation of length:

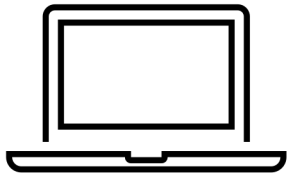
```
length :: [a] -> Int
  length [] = 0
  length (x:xs) = 1 + length xs
```

If we were to evaluate the function call **`length [1,2,3]`** we would get:

```
length [1,2,3]
    = length (1:[2,3])
    = 1 + length [2,3]
    = 1 + length (2:[3])
    = 1 + 1 + length [3]
    = 1 + 1 + length (3:[])
    = 1 + 1 + 1 + length []
    = 1 + 1 + 1 + 0
```

Look at the definition of nth from exercise 4 in this set, and using the same approach as above, show the evaluation of
`nth 5 [9,4,10,1,2,6,9]`

Give it a go!



12. [Hutton, ex 6.4] Define a recursive function `Euclid :: Int -> Int -> Int` that implements *Euclid's algorithm* for calculating the greatest common divisor of two non-negative integers: if the two numbers are equal, this number is the result; otherwise, the smaller number is subtracted from the larger, and the same process is repeated. For example:

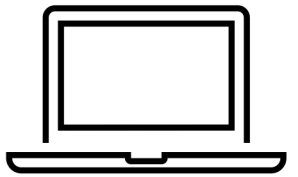
```
> euclid 6 27
3
```

13. [Hutton, ex 6.8] Construct definitions for the library functions that:
    a) calculate the **`sum`** of a list of numbers;
    b) `take` a given number of elements from the start of a list;
    c) select the **`last`** element of a non-empty list.

14. Construct a definition for the library function **`zip`.** This function takes two lists and "zips" them together, producing a list of pairs, such that the first pair contains the first item from the first list and the first item from the second, and so on. If one list is shorter than the other, the function stops when the shorter list runs out of items. For example:

```
zip [1,2,3] ”abcdefg” = [(1,’a’), (2,’b’), (3,’c’)]
```

Give it a go!



15. The Leibniz formula for π is given by:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

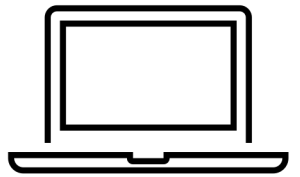
Write a function approx_pi that takes a single argument: the tolerance you want for the return value. The output should be a tuple: the first value in the tuple should be the approximation of the value of π, accurate at this tolerance. The second item in the tuple should be the number of recursive steps that were required to get this accuracy.

For example,

```
> approx._pi 0.001
(3.1420924036835256,2000)
```

(Yes, it converges **very** slowly.)

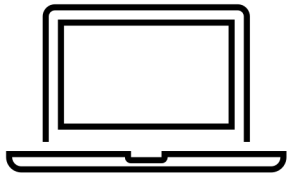
Give it a go!



16. Go back to exercises 1 and 2 in this set. There are actually three ways you could implement each of these:
    a. Using map
    b. Using list comprehension
    c. Using recursion

Provide alternative solutions to exercises 1 & 2 using the two approaches that you *didn't* use the first time around.

Give it a go!



17. [Hutton, ex 5.2] Suppose that a *coordinate grid* of size $m \times n$ is given by the list of all pairs (*x, y*) of integers such that $0 \leq x \leq m$ and $0 \leq y \leq n$. Using a list comprehension, define a function `grid :: Int -> Int -> [(Int, Int)]` that returns a coordinate grid of a given size. For example,

```
> grid 1 2
[(0,0), (0,1), (0,2), (1,0), (1, 1), (1,2)]
```

18. [Hutton, ex 5.6] A positive integer is *perfect* if it equals the sum of all its factors, excluding the number itself. Using a list comprehension and the function `factors` (which you will need to define if you have not already done so in earlier exercises), define a function `perfects :: Int -> [Int]` that returns the list of all perfect numbers up to a given limit. For example:

```
> perfects 500
[6, 28, 496]
```

19. There are a number of library functions that are available to operate upon lists. You have two techniques now for working with lists: list comprehension and recursion. Practice these techniques by writing your own versions of some of the library functions. Good ones to try are: replicate, take, sum, unzip, reverse. Which ones are better suited to a recursive solution? Which ones a list comprehension?