





# Practice problems


 You can use functions developed during the practicals to help you solve some of these more easily 


 If you run into the “Value restriction” error, try using pipelining instead of composition. That sometimes sorts it out!   
(if it doesn't sort it out, WhatsApp me with a screenshot, 082 927 8658, and I'll assist)

## Hints

While none of these hints are essential to completing these questions, remembering them will typically make the problems easier to solve and your solutions easier to write.

1. In the syntax...

**let**  **= value**

...the  is a pattern. You can use this fact to bind values inside an ADT such as a tuple.

2. You can nest patterns to any depth. For example, `(_, a, (b, _))` will verify the structure of a 3-tuple where the last element is a 2-tuple. As another example, given the definition of a **List** on page 3, the pattern **Element** `(_, Element (y, Empty))` will only match lists that have precisely two elements in them; and the pattern **Element** `(x, Element (a, rest))` will match lists that have *at least* two elements in them.
3. Each pattern in **function** syntax is matched from *top to bottom*, and body executed is the body of the first pattern to match.

## Basics

1. Write a function *max* which finds the maximum of two integers.
2. Write a function *abs* which returns the absolute value of an integer.
3. Write a one-line lambda function which only returns true when its input is less than 3.

# Recursion

1. Write a recursive function to find the greatest common divisor (GCD) of two integers using the Euclidean algorithm. The GCD is the largest positive integer that divides both numbers without leaving a remainder. The Euclidean algorithm is an ancient method for finding the GCD of two numbers, based on the observation that the GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number.

You can follow this algorithm:

- a. Take two numbers, **x** and **y**, where **x** is greater than or equal to **y**. If **y** is zero, the GCD is **x**.
- b. Divide **x** by **y** and find the remainder **r**.
- c. Replace **x** with **y** and **y** with **r**.
- d. Repeat steps (b) and (c) until **y** becomes zero. The GCD is the value of **x** at that point.

Treat negative numbers as though they were positive numbers.

2. Write a recursive function in F# to compute the power of a given number. The function should take two integer arguments, **base** and **exponent**, and return the result of raising the **base** to the power of **exponent**.
3. A delivery person has to climb a staircase with a given number of steps to deliver a package. The delivery person can climb either 1 or 2 steps at a time. Write a recursive function to calculate the total number of distinct ways the delivery person can climb the staircase to deliver the package.

The function should take a single integer argument, **steps**, and return the number of distinct ways to climb the staircase with the given number of steps.

4. The square root of a number  $n$  is the value which, multiplied by itself, will result in  $n$ . Square roots are defined for positive numbers only. The Newton-Raphson method finds a square root by guessing a number and then improving the guess iteratively until it is “good enough”. Given a guess  $x_i$  for a number  $n$ , we can say that the next approximation  $x_{i+1}$  can be found using the equation

$$x_{i+1} = \frac{x_i + \frac{n}{x_i}}{2}$$

Create a recursive method that implements the Newton-Raphson method, starts with an initial guess of half the number, and improves that guess through 20 iterations.

# Higher order functions

1. Write a function that takes two arguments: an integer  $n$  and an  $\text{int} \rightarrow \text{int}$  function  $f$ . The function should apply  $f$  to each of the integers from 1 to  $n$  (inclusive) and return a tuple containing the sum of the even results and the sum of the odd results.
2. Write a function that takes two arguments: an integer  $n$  and an  $\text{int} \rightarrow \text{int} * \text{int}$  function  $f$ . The function should apply  $f$  to each of the integers from 1 to  $n$  (inclusive) and return a tuple containing the sum of all the first elements in the tuples returned by  $f$  and the sum of all the second elements in the tuples returned by  $f$ .
3. Write a function that takes three arguments: an integer  $n$ , an  $\text{int} \rightarrow \text{bool}$  predicate  $f$ , and an  $\text{int} \rightarrow \text{int}$  function  $g$ . The function should apply  $g$  to each of the integers from 1 to  $n$  (inclusive) to obtain a result and then filter that result using  $f$ . It should return a tuple containing the number of times the predicate was satisfied and the sum of the results which satisfied the predicate.
4. Write a function that takes two arguments: an input  $x$  and an  $'a \rightarrow \text{int} * 'a$  function  $f$ . The initial value passed to  $f$  is  $n$ . The function should recursively apply  $f$  to the second element of the previous application of  $f$  until the first element of the tuple returned by  $f$  is 1. The function should return the count of applications of  $f$  and the last tuple returned by  $f$ .

# Algebraic data types

The following basic definitions are given to you for use in the following questions:

```
type Maybe<'data> =  
| Nothing  
| Something of 'data  
  
type List<'data> =  
| Empty  
| Element of 'data * List<'data>
```

## Lists

1. Create *last*:  $\text{List}<'a> \rightarrow \text{Maybe}<'a>$ , which extracts the last element of a list, if it exists.

2. Create *init*:  $\text{List}'a \rightarrow \text{Maybe}(\text{List}'a)$ , which returns all elements of a list except for the last one.
3. Create *range*:  $\text{int} \rightarrow \text{int} \rightarrow \text{List}(\text{int})$ , which creates a list of numbers where every element increases by 1. The inputs passed are lower and upper bounds respectively, and both of them should be in the list. For example, **range 3 6** should give you a list containing 3, 4, 5, and 6; and **range 6 3** should give you an empty list.
4. Create *singleton*:  $'a \rightarrow \text{List}'a$ , which creates a single-element list.
5. Create *repeat*:  $\text{int} \rightarrow 'a \rightarrow \text{List}'a$ , which creates a list containing  $n$  copies of the specified value, where  $n$  is the given integer.
6. Create *cons*:  $'a \rightarrow \text{List}'a \rightarrow \text{List}'a$ , which returns a list with one more element at the start of it.
7. Create *append*:  $\text{List}'a \rightarrow \text{List}'a \rightarrow \text{List}'a$ , which appends the second list to the first one.
8. Create *exists*:  $'a \rightarrow \text{List}'a \rightarrow \text{bool}$ , which returns **true** only when the list contains the specified value.
9. Create *all*:  $('a \rightarrow \text{bool}) \rightarrow \text{List}'a \rightarrow \text{bool}$ , which returns **true** only when all elements satisfy the predicate.
10. Create *any*:  $('a \rightarrow \text{bool}) \rightarrow \text{List}'a \rightarrow \text{bool}$ , which returns **true** only when at least one element satisfies the predicate.
11. Write *intersperse*:  $'a \rightarrow \text{List}'a \rightarrow \text{List}'a$ , which places the given value between all elements of the list.
12. Write a **non-recursive** function *concat*:  $\text{string} \rightarrow \text{List}(\text{string}) \rightarrow \text{string}$ , which joins all the strings in the list together, separated by the specified value.  
Hint: you have an **intersperse** function and a **fold** function already...
13. Create *zip*:  $\text{List}'a \rightarrow \text{List}'b \rightarrow \text{List}'a * 'b$ , which creates a list of pairs using elements from the input lists. The list of pairs is as long as the shorter of the two lists.
14. Create *unzip*:  $\text{List}'a * 'b \rightarrow \text{List}'a \times \text{List}'b$ , which does the opposite of *zip*.

15. Create sort: ('a → 'a → bool) → List<'a> → List<'a> which sorts a list according to the provided comparison function. The comparison function returns true only when the first input is less than the second input.  
*Hint: if you had an insert: ('a → 'a → bool) → 'a → List<'a> → List<'a> function which inserted an element into an already-sorted list, creating the sort function would be very easy and trivial...*

## Custom data structures

Create a parametrically polymorphic data type `Tree<'a>` that represents a binary tree. A binary tree is a data structure in which each element (commonly referred to as a node) has at most two children, usually distinguished as the left child and the right child. Each child node is the root of its own subtree, and this hierarchical structure continues recursively. The topmost node in the tree is called the root. The nodes without any children are called leaf nodes. The depth of a node is the number of edges from the root to the node. The height of a tree is the maximum depth of any node in the tree.

(In F#, there is a feature I *haven't* introduced you to that allows you to use comparison operators—in other words, the set { <, >, <=, >=, = }—in parametrically polymorphic code. The language stops you from running code that requires comparison when the things that need to be compared are incomparable<sup>1</sup>.)

I don't know how you're going to define your **Tree<'a>** structure, so I'm not going to provide tests; you'll have to use the REPL and make your own tests to see that you're doing it correctly.

1. Create treeLNR: Tree<'a> → List<'a> which returns the contents of a tree in Left-Node-Right order.
2. Create treeNLR: Tree<'a> → List<'a> which returns the contents of a tree in Node-Left-Right order.
3. Create treeMap: ('a → 'b) → Tree<'a> → Tree<'b>.
4. Create a **non-recursive** function stringLNR: Tree<'a> → string, which returns a string representing the tree in Left-Node-Right order, separated by spaces. You may use the built-in **string** function to convert each node's value to a string.  
*Hint: You already have **concat**, **treeMap**, and **treeLNR**.*
5. Create a **non-recursive** function stringNLR: Tree<'a> → string, which returns a string representing the tree in Node-Left-Right order, separated by spaces. You

---

<sup>1</sup> e.g. images

may use the built-in **string** function to convert each node's value to a string.

6. Create a type **Order** with range 2 and the set of cases { NLR, LNR }. Now create a non-recursive function *stringify*:  $\text{Order} \rightarrow \text{Tree}\langle'a\rangle \rightarrow \text{string}$ , which returns a string representing the tree in the specified order.
7. Create *btAdd*:  $'a \rightarrow \text{Tree}\langle'a\rangle \rightarrow \text{Tree}\langle'a\rangle$ , which adds a value to a binary tree. In a binary tree, the value of each node is greater than or equal to all the values in its left subtree. If the value already exists, no change is made.
8. Create *btExists*:  $'a \rightarrow \text{Tree}\langle'a\rangle \rightarrow \text{bool}$ , which returns **true** only when the specified value exists in the tree.
9. Create *btMax*:  $\text{Tree}\langle'a\rangle \rightarrow \text{Maybe}\langle'a\rangle$ , which returns the maximum value in the tree.
10. Create *btMin*:  $\text{Tree}\langle'a\rangle \rightarrow \text{Maybe}\langle'a\rangle$ , which returns the minimum value in the tree.
11. Create *height*:  $\text{Tree}\langle'a\rangle \rightarrow \text{int}$ , which returns the number of levels between the tree's root and its furthest leaf.
12. Create *btDelete*:  $'a \rightarrow \text{Tree}\langle'a\rangle \rightarrow \text{Tree}\langle'a\rangle$ , which removes the specified value—if it exists!—from the specified tree. Note that when the tree has two children, you may need to replace the specified value with either the maximum of the left subtree or the minimum of the right subtree.