HLP Worksheet 2: Working with Collection Data

Contents

Prerequisites

You should be familiar with the F# elements introduced in Tutorial 1 and the Introduction to Tutorial 2 from Lecture 2. As always the new F# in this worksheet (summarised in the yellow box) will be learnt going through the worksheet - don't try to learn it in advance!

Key to coloured boxes in these Worksheets

F# Language Reference

Syntax	Name	Description
match *e* **with** | *pattern1* **->** *e1* | *pattern2* **->** *e2*	match expression	Match *e* against one or more patterns, returning the expression for the first matching pattern.
x.[n]	lookup	lookup array index **n** in array **x** lookup key **n** in map **x**.
let mutable x = *e*	mutable value	**x** is initialised to value e
x <-*e*	assignment	set mutable value or array element **x** to value *e*

Figure 1. F# elements

Match patterns
In F# match patterns are very powerful.

Pattern	Meaning
_	wildcard
(x,y) **as** tup	named pattern binds **x**,**y** and **tup**
literal	constant to match
pattern **when** *exp*	match requires *exp* to be **true**

Match Gotcha

You can't match part of a pattern to an identifier, because all identifiers in match patterns are taken to be variables that are bound to the pattern, as in line 2 below. Line 3 shows the correct way to do this.

```
1: match x with
2:    | (1,v) -> // v is set to snd x
3:    | (2,v') when v = v' ->
4:          // matches when snd x = v
```

Match Example

Pattern matching can be used with tuples (as in the example) lists, and any other non-function type.

```
1: let testMatch x =
2:    match x with // Patterns are matched in order till a match is found
3:    | (2,a,_) -> a*a // 3-Tuple starting with 2. The 2nd value is don't care and bound to x.
4:                     // The third value is don't care and not used
5:    | (a,0,0) as y -> myFunc a y // Tuple of form (_,0,0); a and y are bound. y is bound
6:                                 // to whole tuple
7:    | (0,0,n) when n < 0 -> n // Tuple with n < 0 and other parts 0
8:    | (1,2,a ) | (a,4,3) -> a // Two matches with a single result expression
```

```
 9:     | (x,y,_) -> x+y // Any tuple not previously matched. Note that x,y are bound here
10:                     // to 1st and second components of tuple so this is
11:                     // NOT the original x (better to use a different name)
```

Curried functions: recap

F# uses different syntax from most other languages for applying functions to parameters. This is surprising when you first meet it, but makes functional code easier to read by reducing brackets. Check that you now understand this from the following examples:

C	F#	Notes
`f(x)`	f x f (x)	function application does not need brackets.
`f (g(x))`	`f (g x)`	`f g x` will be parsed as `(f g) x`
`printf("fib(%d)=%d\n",n,fib(n))`	`printfn "fib %d=%d" n (fib n)`	`printfn` takes *curried* parameters

Learning outcomes

In this Workshop you will learn to use both recursion and standard library functions to manipulate data in lists, arrays and maps. You will be able to select an appropriate collection type (list, array, map or set) for any specific application. You will be able to choose a good implementation (recursive or non-recursive using standard library) for any specific data transformation problem. You will be comfortable writing small functional fragments of code, and using type information to debug these.

Exercise 1. Lists

This section of the tutorial will show you how to transform data in lists. In the process you will learn some new FSharp match expression and list syntax, how to write recursion in Fsharp, and the merits of recursive and non-recursive implementations.

Reference material on F# lists

- F# List type overview
- F# List module functions documentation

1.1 List implementations

The notation `[ a ; b ]` for a list is a shorthand for the list's construction using binary list nodes and a terminating `[]` (the empty list `List.empty`). A binary list node with head `a` and tail (rest of list) `lst` is written `List.Cons a lst` or (infix equivalent) `a :: lst`.

Q1. Read the Fsharp documentation on operators for working with lists `::`. Using only:

- The `List.Cons` binary operator: `::`
- The `List.empty` constant: `[]`
- Numeric literals

Construct explicitly the list `[1;2;3]`, checking with FSI. Does `::` associate to the left or right? Why is this helpful?

Answer

Q2. Run `fsi` and determine approximately (to within a factor of 2) how long a list of the form `[1..n]` can be constructed, e.g. a 10,000 element list is:

```
1:  [1..10000]
```

Does processing (via a pipeline) with `List.map (fun n -> n+1)` alter the size of list that can be processed?

Answer

Q3. There is a 64 bit version of fsi named `fsianycpu` which you can run from the start menu. Determine how much longer lists can be before running out of space using this? The limits in both these cases come from the process memory space limits for a Windows .NET executable.

Windows .NET space limits

Answer

Q4. For a 32 bit implementation you would expect at least 8 bytes per list element, and probably 12. Why? How much for a 64 bit implementation? (Look at the answer if you can't do this).

Answer

Key to coloured boxes in these Worksheets

```
1:  let rec map1 f lis =
2:      match lis with
```

```
3:     | [] -> []
4:     | h :: tl -> (f h) :: map1 f tl
```

The function **map1** is a recursive definition which implements the map function and introduces some new F# syntax.

- The **rec** keyword is required syntax after **let** in any recursive function to allow the function itself to be used in its own body.
- The **match e1 with ...** expression is an F# equivalent of **switch**. Each vertical bar introduces a new match case of the form **pattern -> expression** and the first case for which the *pattern* matches delivers the expression result as the value after the **->**. Note that the vertical bars must align with the **match** keyword.
- In this code the two cases cover the two possibilities for a linked list, which is either **[]** or of the form **x :: y** where **x** is the head of the list and **y** is the tail.
- In this code note that **h** and **tl** are arbitrary new identifiers: they don't have to be **h** and **tl** as here. These identifiers are bound to the corresponding parts of the list in the match expression to the right of the **->**. The match expression simultaneously gives conditional execution and data decomposition into parts.

In the second case **::** is used to decompose, or put together, a list:

- **h :: tl** this **::** in a *pattern* decomposes the list into its first element bound to **h** and the rest of the list bound to **tl**
- **(f h) :: ...** this **::** operator creates a new list constructed from **(f h)** as first element and **...** as the rest of the list. This makes the function call **f** applied to **h** the first item of the result list. The rest of the result list comes from **...**
- **map1 f tl** this recursive call as **...** makes the rest of the *result list* the map of **f** over the rest of the *input list*
- this is a *recursively defined function* typical of list processing. This template using **::** to separate the first part of the list from the rest is often used.
- each recursive call of **map1** will apply **f** to one element of the input list and then make a recusrsive call to implement the rest.

Q5. A common mistake is to forget the *end case* of a recursion. In **map1** delete the **[]** pattern match. What errors are shown in the editor before running the code?

Answer

In F# can we implement user-defined recursion without this stack limit where needed by using *tail recursion*. A tail recursive function has all of its recursive calls written in such a way that no local information needs to be saved after the recursive call, equivalently the recursive function call value is the value returned from the function without any subsequent processing. In this case the recursive call can be compiler optimised into a branch with no stack use.

Q6. Is **map1** *tail recursive* or not? Why?

Answer

Q7. What happens with larger lists (e.g. 20,000 items)?

Answer default stack memory limit is 1MB - to keep Garbage Collection fast

The trick to using tail recursion is to put all the intermediate data into parameters of the recursive function.

```
1:  let map2a f lis =
2:     let rec map2a' ret lis =
3:        match lis with
4:        | [] -> ret
5:        | h :: tl -> map2a' (f h :: ret) tl
6:     map2a' [] lis
```

Some points to note about this definition:

- The inner function **map2a'** is needed because it must be called (last line) with the correct initial value **[]** for the **ret** parameter.
- In the inner function **f** never changes and therefore need not be a parameter. The outer **f** has scope the inner body.
- In F# **'** is allowed in identifiers, it is conventionally used here as **map2a'** to indicate an inner function with similar meaning to the outer.

Q8. Run **map2a** as defined above on the list **[1..10]** with the function **fun x -> x*x**. What do you notice?

Answer

Q9. The most obvious way to avoid the list ordering problem in the last answer **map2a** would be to replace **(f h :: ret)** by **(List.append ret [f h])** or equivalently using the append operator **(ret @ [f h])**. The square brackets turn **f h** into a list with one element equal to **f h**. Write **map2a** modified like this as **map2b** and check that it is now a correct implementation of **map**.

Answer

The *time order* of a function (in a functional language) is determined by counting the total number of recursive calls for given size of input, since each recursive call invokes the function body once - not including its own recursive calls. Time order is an asymptotic measurement. Most list functions recurse once for each element of the list and therefore have time order $\Theta(n)$[1].

Q10. What is the *time order* of **map2b** and **map2c** from the Q9 answer? Look at the answer to this question if you are not sure how to work this out.

Answer

In `fsi` you can enable timing with the command `#time "on"` . Check that `map2c` will work with long lists as expected.

Q11. Compare `mapTailRecRev` and `mapTailRecAppend` approximate speed on large lists with the library `List.map` function. Is this what you expect?

```
let mapTailRecRev f lis =
   let rec map2c' ret lis =
      match lis with
      | [] -> List.rev ret
      | h :: tl -> map2c' (f h :: ret) tl
   map2c' [] lis

let mapTailRecAppend f lis =
   let rec map2a' ret lis =
      match lis with
      | [] -> ret
      | h :: tl -> map2a' (List.append ret [f h]) tl
   map2a' [] lis
```

Answer

Exercise 2. List library functions

List.fold

We have already seen how `List.map` and `List.reduce` can implement iterative operations on lists without any user-defined recursion. An additional benefit is that the library implementation of all these functions will be iterative and highly efficient, with no stack use.

`List.fold` is a more general version of `List.reduce` which operates on a list using a function `folder` and a supplied initial value `initState`. Thus `List.fold f init [a1;a2;a3...an]` is calculated as:

```
f (f (f (f init a1) a2) a3 ...) an
```

```
let plus = (+) // NB this is a "functionised operator"
let adder1 = List.fold plus 0
let adder2 = List.reduce plus
```

Here with `initState` set to `0`, `adder1` and `adder2` are nearly identical, since:

```
adder1 [1;2;3] = plus (plus (plus 0 1) 2) 3)
adder2 [1;2;3] = plus (plus 1 2) 3
```

Q12. What input list will work with `adder1` but not with `adder2`?

Answer The empty list will lead to an error with adder2, since List.reduce requires at least one element in the list. It will work fine with adder1 returning 0.

So far we have not said much about how you can make the functions you pass to `List.fold` etc. In the above example the correct function (`plus`) is given to you as a functionised operator `(+)`. What if you need a user defined function?

1. You could define this function in a let block with some suitable name `myStrangeFunc` and use it: `List.fold myStrangeFunc []`, as is done above where `plus` is defined in a let block.
2. You could write the new function as an anonymous function.
3. You could use a curried function with one of its parameters partially applied as in `(+) 1` which is the same as `plus 1` and therefore `fun x -> x+1`.
4. You could write a high order function that returns another function as its value. Such a (function) value is a called a *closure* for the reason explained below.

These are all equally valid. F# as a functional language makes it very easy to define functions. When defining closures there is however an extra complication. It is necessary to understand the way that local variables are used in the function that defines the closure:

```
let n = 17
let makeIncrementer x =
    let n = 2*x
    let incr i = i+n
    incr // incr is a closure with bound n
let f1 = makeIncrementer 10
let f2 = makeIncrementer 20
printfn "n1 = %d" n
let n = 3
printfn "f1 1 = %d" (f1 1)
printfn "f2 3 = %d" (f2 3)
printfn "n2 = %d" n
```

Q13. Work out what the print statments will print out. What does this tell you?

Answer

`List.fold` is also more useful than **`List.reduce`** because the type of its **`accumulator`** parameter can be anything, and different from that of the list elements, as shown in this example:

```
1: List.fold (fun n s -> (String.length s) + n) 0 ["a";"bc"; "def"] // computes the total number of characters in a list of strings
```

List.fold slides

Q14. Implement a function using **`List.fold`** that returns the length of a list. (Equivalent to **`List.length`**).

Answer

Q15. Implement list reverse using **`List.fold`** and the *add the head onto the front of an accumulator parameter* method used in **`rev1`** in (@?rev). The beauty of this is that **`List.fold`** does the recursion and you need only write a **`folder`** function to make the accumulator change.

Answer

Q16. Implement a function **`listSumOfSquares`** using **`List.fold`** that returns the sum of squares of elements in an **`int`** list. What needs to change for it to work with **`float`** lists?

Answer

Other Library Functions

Debugging type errors

Very soon, if not already, you will get annoyed by the fact that your programs **do not type check** and you can't work out why. **`List.collect`** used below tends to create problems. You must make types something you can always (precisely) work out, and debug methodically. Luckily the compiler gives you a lot of help if you have the correct strategy. When presented with a type error not immediately obvious simplify as follows:

1. Turn any anonumous functions into named functions, check the type of their name
2. Label functions with the type of parameters: **`f x y = ...`** -> **`f (x: int list) (y:int list list) = ...`**
3. If needed work out the type of any subexpression by isolating it as a let definition and checking the type of the defined value. Since let definitions can occur before any expression this is always possible, and will simplify complex expressions where this is needed to help understanding.

Q17. implement **`List.filter`** using **`List.collect`**.

Answer

Q18. Using **`List.indexed`**[2] and **`List.filter`** and the remainder operator **`%`** , write a function **`listOdd`** that returns the odd elements of a list (counting from 1, not 0 as in indexes!)) without the even elements. Also a similar function **`listEven`**.

Answer

Q19. Alternatively, avoid all use of indexing (which tends to generate errors), and implement **`listEven`** by chunking a list into 2 element sublists (**`List.chunkBySize`**) and then collecting (why not **`map`**?) the first element of each sublist. Use a **`match`** statement, and note that you need to be careful to implement the 1 element sublist case that will happen when the original list has an odd length. One common language issue when matching lists is that fixed length list matches (as used here) will always result in a warning because they are incomplete. You can fix this with a "catch-all" match pattern leading to a *failure*: an exception that terminates execution with an error message. The **`failwithf`** function, which operates like **`printf`**, is useful. Never leave "pattern match incomplete" warnings without an informative error message. For example:

```
1:     match list with
2:     // other matches
3:     | _ -> failwithf "List of length > 2 (%A) should never happen" list
```

Answer

Q20. Using **`listOdd`** and **`listEven`** and **`List.zip`** write a function **`listPair`** that returns a list each element of which is a pair of successive elements of the input. Repeat the last element if the input list has an odd number of elements.

```
1:  listPair [1;2;3;4] = [ (1,2) ; (3,4) ]
2:  listPair [1;2;3;4;5] = [(1,2) ; (3,4); (5,5)]
```

Answer

Q21. Implement **`listPair`** with the method of Q19 using **`List.chunkBySize`**, **`List.map`**, and a suitable **`map`** function that uses **`match`**. Note that match patterns can be fixed length lists, tuples, or contain wildcards:

- **`|[ a ; b] ->`** matches two element lists with **`a`**, **`b`** bound to the two elements
- **`| (a,b) ->`** matches the 2-tuple **`(a,b)`** binding **`a`** and **`b`** to its two components.
- **`| _ ->`** matches anything
- **`| [(a,b) ; (c,_)] ->`** matches a list of exactly two tuples, binding items **`a`**, **`b`**, **`c`** as shown.

Answer

Q22. Implement **`listPair`** using a recursive function (which need not be tail recursive).

Answer

Q23. Using **`List.groupBy`**[3], **`List.map`** and **`List.length`** Implement a function **`listHistogram`** that returns a list of pairs (value, frequency) when given a list of integers:

```
1:  listHistogram [ 8; 1; 2; 7; 2; 1 ] = [ (1,2) ; (2,2) ; (7,1) ; (8,1) ]
```

Answer

Exercise 3: Taylor Series Revisited

Workshop 1 showed you a highly inefficient way to implement a Taylor series where the code looked very like the mathematical equation. If we want a more efficient implementation, how can we do it?

The obvious way to make a Taylor series calculation efficient is to compute each term from the previous one. For the **`sin`** series:

$$T_{n+1}(x) = -\frac{x^2}{2n(2n+1)} T_n(x)$$

$$T_0(x) = x$$

$$\sin x \approx \sum_{i=1}^{N} T_i(x)$$

We could implement this as EITHER a **`List.fold`** over the list **`[1..N]`**, where the fold **`state`** is a tuple **`(sumOfPrevTerms,nextTerm)`** OR using a (local to the main function) tail recursive function with three parameters: **`i`**, **`sumOfPrevTerms`** and **`term`**. It should be now be obvious how this works, in both cases.

Q24. Write the two implementations. Check your answers against those given.

Answer

Reflect on this. Is there any advantage of the F# tail recursive implementation over an imperative language?

Q25. Suppose you had to write a lot of Taylor series functions. Could you write a high order function (a variant of **`List.fold`**) to make this easier?

Answer

Exercise 4: Sieve of Eratosthenes

The sieve of Eratosthenes is a famous and very old algorithm for obtaining prime numbers. See here for an animation. In this section we will contrast different F# implementations using arrays and lists, learning the relevant F# syntax as we go.

As often presented:

1. Create an array **`sieve`** of **`N`** booleans indexed **`2..N`** all initialised to true. Index **`i`** will be calculated true or false according to whether **`i`** is prime.
2. **`p :=`** {lowest **`i`** such that **`sieve[i] = true`** }. Output **`p`** as the next prime, or stop if **`p`** does not exist.
3. Forall integer **`i`** such that: **`i >= p*p`** and **`p*i <= N`**: **`sieve[i] := false`**. (Multiples of p less than p^2 will also be factors of some prime smaller than p and so need not be considered).
4. Goto 2.

We will explore this algorithm, and F# arrays, together. As we look at different implementations we will think about to what extent each one is "functional" and how this impacts clarity and efficiency.

Standard F# type *array* implements arrays of data, and is also a standard .Net type, used as input or output from .Net ecosystem functions often called from F#. It represents arrays of *mutable* elements. As far as possible syntax and operations (map etc) are uniform with F# lists:

```
1:  histData = [| 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 |] // as with lists, note [| |]
2:  histData = Array.zeroCreate 10 // better way to create a large array
3:  primeSieve = Array.create 10000 true // an array initialised to any specific value
```

In F# math functions and constants are found in *class* Math (not module Math!). Thus π is referenced as **`Math.PI`** and the **`Math`** prefix cannot be omitted by opening a module. Square root is built in to the language as **`sqrt`** (as well as being found under **`Math.Sqrt`**).

```
1:  open System // for Math.Ceiling
2:  let primes upTo =
3:     // array of booleans. Index 0 and 1 are not used.
4:     let primeSieve = Array.create (upTo+1) true
5:     // Largest prime factor we need to sieve by
6:     let maxPrimeFactor = upTo |> float |> sqrt |> Math.Ceiling |> int
7:     // Return array of primes from the current value of primeSieve
8:     let sieveToPrimes =
9:        Array.indexed
10:       >> Array.collect (fun (i,b)-> if i >= 2 && b then [|i|] else [||])
11:    // set elements with indexes multiples of p to false in primeSieve array.
12:    let removeMultiples p =
13:       if primeSieve.[p]
14:       then [| p*p..p..upTo|]
15:            |> Array.iter (fun p -> primeSieve.[p] <- false)
16:    [|2..maxPrimeFactor|] |> Array.iter (fun p -> removeMultiples p)// do all the prime factor sieving
17:    sieveToPrimes primeSieve
```

New F# syntax

- **`primeSieve.[p] <- false`** The **`.[index]`** notation denotes an array element, and assignment (only allowed on mutable objects in F#) is **`<-`**.
- **`p*p..p..upTo-1`** indicates a range with initial value **`p*p`**, increment **`p`**. In this case, inside **`[| |]`**, it specifies an array of **`int`**.
- **`Array.iter`** Like **`Array.map`** but returns **`()`** and useful where the applied function is used for its *side effect*.
- **`>>`** function composition:
  - **`f >> g`** represents the function made by applying first **`f`** and then **`g`**: **`fun x -> g (f x)`**
  - **`(f >> g) x`** is therefore the same as **`x |> f |> g`**.

Q26. Rewrite line 9-10 using **`|>`** instead of **`>>`**

(Q26 Answer not supplied)

Q27. Simplify line 16.

Answer

Q28. **`Array.map`** applies its function to all array elements like **`Array.iter`**. Rewrite line 16 using **`Array.map`**. Why is this is not a good idea?

Answer

Beware of F# array syntax

F# uses **`[| ... |]`** for arrays and **`[ ... ]`** for lists. However .NET follows C and uses **`int[]`** to mean an *array* of **`int`**. One common type error is to use **`list`** when **`array`** is needed or vice versa. When you read the type errors you must remember that **`int []`** is an *array* of int not as might be expected a *list* of **`int`**. The syntax here makes this mistake very likely!

This implementation shows the strength of F# in handling mixed functional and imperative code. The arrays element assignments are "harmless" in the sense that the result is independent of the order in which they are made. That makes reasoning about this code much easier than is typical of imperative code. The code is still difficult to read. The data is initialised on line 4, mutated on line 16, and output on line 17. Using side-effects makes this possible.

Line 17 can be rewritten as a pipeline of functions with side effects which transforms the initial value of the sieve, making the program structure clearer. The initialisation in line 4 naturally becomes the start of the pipeline. This shows how, even when programs have side effects, these can be packaged in a way that makes them clear, and the program easy to understand.

Q29. Rewrite the function in this manner.

Answer

Q30. The pipeline functions here could just as well be written without side-effects, each returning a modified verion of the sieve array. To do this you would want an efficient *immutable array* data structure with an element update operation that returned a cppy (like Map). In fact .NET does have such a data structure, under **`System.Collections.Immutable`**. Find this and read the documentation. Consider whether they would make a good alternative in this problem. (You can use such non-core functions if you install via NuGet in VS or if you download the github code, compile it, and reference the dll in VSCode).

Answer This is one of those problems where a pure functional solution, using immutable arrays, is less efficient. The array update, even though implemented efficiently, still scales with logN where logN is the size of the array. Also array read scales as logN. That makes this significantly slower than the implementation with side effects.

The array implementation is easy to read but quite long. Let us try for a shorter implementation using lists by restating the algorithm as an inductive function.

Suppose **`sievedBy n`** is a function that filters a list of numbers so that all multiples of $i : 1 < i < n$ are deleted. Thus: **`sievedBy 10`** has multiples of 2,3,5,7 deleted (4,6,8,9 are therefore also deleted).

It is easy to prove that:

- If **`sievedBy n [1..m]`** is not empty, when n is a prime, its first element is the smallest prime greater than n, since none of its elements can have divisors $< n$.
- If the ith prime is p_i and $2 < p_i \leq m$ then p_i = head (**`sievedBy`** p_{i-1} **`[1..m]`**)

This leads to a recursive definition of the list of primes p_i:

```
1:  let rec filterNonPrimes lst =
2:      match lst with
3:      | (p::xs) -> p :: filterNonPrimes (List.filter (fun x -> x % p > 0) xs)
4:      | []      -> []
5:
```

```
6:  let primes n = filterNonPrimes [2..n]
7:  printfn "%A" (primes 50)  // [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47]
```

The previous algorithm processes every multiple (beyond p^2 of every prime p). This algorithm processes every number not divisible by primes less than p to see whether it is a multiple of p.

Q31. Rewrite **`filterNonPrimes`**, giving the anonymous function (**`fun ...`**) a suitable name. This intermediate value *labelling* is always a judgement call. Write down the pros and cons in this example. What are the general principles that determine when you should break up an expression in this way?

Answer

Q32. This functional implementation is not tail recursive, and therefore not useful for determining large prime numbers. Transform it into tail recursive form and calculate all primes in the range $[1, 100000]$.

Answer

Q33. (Optional) The work done by **`filterNonPrimesTR`** is more than **`primes`** because we are checking many $i < p^2$ for whether they are divisible by p. This is not required, all $i < p^2$ which are divisible by p will also by divisible by some prime $< p$. This is the optimisation incorporated into step 3 of the Eratosthenes Sieve algorithm. Make a highly optimised version of **`listOfPrimesTR`** that can generate very large lists of primes.

Answer

Exercise 5. Maps and Memoise

In this exercise we will illustrate the use of F# *map* types by writing and using a *memoisation function*.

In all high level languages the *map* type (also known as *dict* or *hash* in other languages), representing a data structure that implements dictionary lookup from keys to values, is one of the most useful building-blocks for program data. Maps are implemented with efficient hash-based lookup tables managed entirely by the language standard library functions.

Maps in F#

A map or dictionary type has two type variables: the *key* type (on which lookup is based) and the *value* type. In F# the key type is restricted to be one that can be compared and tested for equality. Scalar types, lists, maps, tuples, records are valid key types. Functions are not. Arrays are valid but should only be used if they are never mutated to avoid confusion.

Dictionaries are created by applying the **`Map`** function to a list or array of (key,value) pairs. Key operations on maps are *lookup* (same as arrays, you can think of maps as immutable arrays with more complicated indexes), and *update* (**`Map.add`**). The keys in a dictionary must be distinct, the values can have duplicates. Dictionary functions are in the **`Map`** module, e.g. **`Map.map`**. Note that **`Map`** (dictionary) has no connection with **`map`** (mapping function) as in **`List.map`**,**`Array.map`** and **`Map.map`**. We often (loosely) refer to dictionaries of type **`map`** as maps.

> There are two ways in F# to write type constructors. For example: **`list<int>`** and **`int list`** are two ways to refer to a **`list`** that has a single type argument **`int`**. The latter form is conventionally used only with built-in F# types such as **`list`** and **`option`**. If there are multiple type arguments, you normally use the syntax
>
> ```
> 1:  map<int, string>
> ```
>
> but you can also use the syntax **`(int, string) map`**.

In F# maps, unlike arrays, are **not mutable**. That means that **`Map.add`** *cannot change the existing dictionary*, instead it returns a new dictionary with the specified key value added or updated.

```
1:  let xm = Map ["first","the" ; "second","cat"]
2:
3:  let ym = Map ["cat",7 ; "the",3]
4:
5:  let zm = Map [ [1;2;3],5 ]
6:
7:  zm.[[1 ; 2 ; 3]] // look up key in zm map (returns 5)
8:
9:  let ym' = Map.add "fox" 5 ym // new map with ("fox",5) added to ym.
```

Q34. What is **`ym.[xm.["first"]]`**?

Answer

Q35. What type are **`xm`**, **`ym`** and **`zm`**?

Answer

Q36. How would you obtain the list of all keys in a map **`m`**?

Answer

Prime factorisation

Given a positive integer n, return the list of its prime factors. This is a time-consuming operation, on the difficulty of which most modern public-key cryptography is based. The brute-force method is to test repeatedly for possible factors.

Find prime factors of integer n

1. $c := 2$
2. if c divides n then return c appended to the prime factors of n/c
3. $c := c + 1$
4. If $c * c > n$ then n is prime, so its own sole prime factor, else goto 2.

We will return the complete list of prime factors, allowing duplicates, in ascending order, so that the result for $n = 12$ would be `[2,2,3]`.

Q37. Write this as an F# function `primeFactors n` with the iteration incrementing c in steps 2-4 (which may be very long) tail recursive. The recursive function call in step 2 need not be tail recursive since the depth of recursion here is low.

Answer

Q38. The *depth* of recursion is the maximum number of nested recursive calls during execution. What is the maximum depth of non-tail recursion in your answer to the previous question if n is a 32 bit signed integer? What is the maximum total depth of recursion?

Answer

This has non-tail recursion depth = number of factors and therefore worst case log2n.
. The tail recursive depth is roughly n.
. However, tail recursion is optimised to a loop by the F# compiler, in which case the depth will 1.

Memoise

We will speed this function up by remembering all previously calculated results in a map. If the input n has been previously used the correct answer is returned immediately, otherwise the calculation is done, and the result is returned, *and also stored for later use*. This is called *memoisation*.

Memoisation is a powerful technique to speed up recursion, in particular. Ironically it is not strictly functional. We need to wrap `primeFactors` inside a function that has a *mutable map* as a local variable. Since we are using F# *immutable* maps we need instead to have a mutable local variable, whose value can change. Mutable variables are exactly what you have used throughout your life programming in imperative languages.

The type of the memoise function is:

```
val memoise : fn:('a -> 'b) -> ('a -> 'b) when 'a : comparison
```

Thus it takes a single parameter which is a function `'a -> b'` and returns a function of the same type. In addition the type `'a` must support comparison (so that it can be used as a map key).

The syntax for declaring a mutable variable in F# is `let mutable` and for assigning to it, as with arrays, use the operator `<-`. Mutable variables behave just like variables in an imperative language.

```
let mutable x = 1
// function with side effect
let addOneToX() = x <- x+1
addOneToX()
addOneToX()
```

```
let memoise fn =
   let mutable cache = Map []
   fun x ->
       ....

let square x =
   printfn "Square called with x = %A" x
   x*x

let mSquare = memoise square
mSquare 10
mSquare 3
mSquare 10 // result is in cache and not recomputed
let mSquare1 = memoise square
mSquare1 10 // result is computed again because mSquare1 has a different cache from mSquare
```

There is an important issue here of variable *scope* and *lifetime*. The variable `cache` is initialised and bound during the call to `memoise` and the binding will be in scope in the anonymous function `(fun -> ...)` that is returned. Every time this function is called with a new parameter, a new $(key, value)$ will be added to map `cache`. The lifetime of `cache` is the same as that of the returned function.

Compare this with a structure in which the `cache` variable is declared and initialised inside the returned function:

```
let memoiseBad fn =
   fun x ->
      let mutable cache = Map []
      ....
```

`MemoiseBad` will not remember anything because `cache` is initialised every time the returned function is called.

Q39. write and test the function `memoise`.

Answer

Ticked Exercise

Write a function `findLargestPrimeFactor: int64 list -> int64 list`. This will find, for each number in the input list, the corresponding *largest prime factor*. The output list is of the largest prime factors in the corresponding same order. You may assume that no input number is larger than 2^{40}. Your code should work efficiently for long lists.

The type `int64` represents 64 bit integers. An `int64` literal is written as the sequence of digits followed by `L`. For example: `1234561002341L`. As expected `int64` converts other numeric types into `int64`, `float` will convert `int64` into `float`, etc.

For very large integers F# provides `bigint` (an alias for `BigInteger`). You do not need to use this.

Submit your function definition without test code as an `fsc` file to the Worksheet 2 Tick link.

Extension Questions

E1. Modify the prime number factorisation code to use `bigint` and test it on some big numbers. What fraction of numbers, of roughly 2^{100} size, can be tested for primality in a reasonable time? To answer this question properly you would want to explore the use of *asynchronous* parallel computation in F#, a topic for much later.

E2. F# can be taught as a self-contained simple functional language - which is how you will learn it. When needed F# can use (with no problems) the entire .NET ecosystem of libraries. Relevant to this tutorial is the available collections. There are many variants, e.g.:

- Immutable arrays
- Mutable maps
- Very efficient extendable arrays (like python lists)

Find out about these. Are they better than the built-in F# collections? Why do you think I don't teach using the available immutable array collection - which would make more sense when starting with a pure functional language?

E3. Is there a much better way to factorise large numbers than the brute force method used here? What will happen to the world's cryptographic systems if/when such a method becomes viable (It has been considered)?

Reflection

- This worksheet has introduced two ways in which F# is *not* a pure functional language: mutable arrays and mutable variables. How much should a (good) F# programmer use these features? are they really needed at all?
- If you want to have a tough time, you can program in F# is a normal imperative language. Even loops are available in F# if you want (though they are not explicitly taught in this module and seldom used). Find out how to write F# loops. Do loops have any use without mutable variables or arrays?