Student Name: Your name, your.email@ntut.edu.tw
Student ID: Your student ID number,
Instructor: Jyun-Ao Lin, jalin@ntut.edu.tw

Exercise 1: Number warm-up

Write the following functions in OCaml

(a) `fact` such that `fact n` returns the factorial of the positive integer `n`.

(b) `nb_bit_pos` such that `bn_bit_pos n` returns the number of bits equal to 1 in the binary representation of the positive integer `n`

Exercise 2: Fibonacci

Here is a naive writing of a function calculating the terms of the Fibonacci sequence

```
let rec fibo n =
  (* precondition : n >= 0 *)
  if n <= 1 then
    n
  else
    fibo (n-1) + fibo (n-2)
```

Write a new version of this function with linear complexity in the parameter `n`.

Hint. You can use an auxiliary function using two accumulators.

Exercise 3: Strings

Writing functions in OCaml

(a) `palindrome` such that `palindrome m` returns `true` if and only if the string `m` is a palindrome (that is, we see the same sequence of characters whether we read it from left to right or from right to left, e.g. *madam*)

(b) `compare` such that `compare m1 m2` returns `true` if and only if the string `m1` is smaller in lexicographic order than the string `m2`
(that is, `m1` would appear before `m2` in a dictionary);

(c) `factor` such that `factor m1 m2` returns `true` if and only if the string `m1` is a factor of the string `m2`
(that is, `m1` appears as is in `m2`).

Exercise 4: Merge sort

The merge sort algorithm sorts a list by applying the following principle:

(i) cut the list into two roughly equal parts;

(ii) recursively sort each of the two obtained lists;

(iii) merge the two sorted lists while preserving the order.

Write the functions

(a) `split` such that `split l` returns two lists obtained by sharing the elements of the list `l` in a manner as balanced as possible;

(b) `merge` such that `merge l1 l2` returns a list containing the elements of the lists `l1` and `l2` sorted in ascending order, assuming that each of the lists `l1` and `l2` passed as a parameter is itself sorted in ascending order;

(c) `sort` such that `sort l` returns a list containing the elements of the list `l` sorted in ascending order.

Exercise 5: Lists

Write the functions

(a) `square_sum` such that `square_sum l` returns the sum of the square of the integers in the list `l`.

(b) `find_opt` such that `find_opt x l` returns `Some i` if the element `x` appears in the index `i` of the list `l` (but not before), and `None` if `x` does not appear in the list `l`

Redo the exercise without using the keyword `rec`. To replace it, use and abuse the functions in the OCaml library `List`.

Exercise 6: Tail recursion

Create a list `l` containing in order the positive integers from zero to one million. Then write functions `rev` and `map` corresponding to the functions `List.rev` and `List.map` from OCaml.

You will need to make these functions applicable to the previous list `l` without causing a stack overflow.

Bonus. Rewrite the functions from the previous exercises to make them tail recursive, if relevant.

Exercise 7: Concatenation

Here is a way to code the concatenation of two lists in OCaml.

```
let rec concat l1 l2 = match l1 with
| []   -> l2
```

```
    | x::s -> x :: (concat s l2)
```

This function, like the `@` operator provided by OCaml, has a cost proportional to the length of the first list. In order to be able to perform multiple concatenations without fear of their cost, we propose a new representation of sequences, based on the following data type (which can be seen as a concatenation tree).

```
    type 'a seq =
    | Elt of 'a
    | Seq of 'a seq * 'a seq
```

The concatenation of two sequences `s1` and `s2` is therefore simply `Seq(s1, s2)`. We can give ourselves a writing shortcut `s1 @@ s2` with the definition

```
    let (@@) x y = Seq(x, y)
```

Such a tree represents a sequence, obtained by considering all its elements in order from left to right. Both trees `Seq(Elt 1, Seq(Elt 2, Elt 3))` and `Seq(Seq(Elt 1, Elt 2), Elt 3)` are the two possible representations of the list `[1; 2; 3]`.

Write the following functions for this sequence structure:

(a) `hd, tl, mem, rev, map, fold_left, fold_right` corresponding to functions of the same name on lists;

(b) `seq2list` such that `seq2list s` returns a OCaml list representing the sequences (do not use `@`);
Bonus (difficult): give a tail recursive version of this function;

(c) `find_opt` such that `find_opt x l` returns `Some i` if the element `x` appears at index `i` in the sequence represented by `i` (but not before) and `None` if `x` does not appear in `s`;

(d) `nth` such that `nth s n` returns the index element `n` in `s` (and throws an exception if the index is not suitable).

How to enrich our sequence structure to potentially make the function `nth` more efficient?

Define the corresponding new type and redefine the functions `(@@)` and `nth` accordingly. Are there any other functions that still need to be updated?