Important notes about grading:

- **Compiler errors:** In cases where programs cannot be successfully compiled, a grade of zero will be assigned. If you encounter difficulties in compiling your assignment, we encourage you to seek assistance by asking questions on Piazza, attending recitation sessions or consulting during office hours.
- Feel free to declare any function recursive by having the `rec` keyword in its function header.
- Your solutions to earlier functions can be used to aid in writing later functions.
- **You can always add a helper function for any of the functions we ask you to implement, and the helper function can also be recursive**.
- You may rename arguments however you would like, but do not modify function's name. Doing so will cause you to fail the function's tests.

Compile and run your code:

After downloading and unzipping the file for assignment1, in your terminal, `cd` into the directory which contains `src`. You should write your code in `src/assignment1.ml`.

Use `ocamlc -o test src/assignment1.ml` to compile the code. Then, use `./test` to excute it.

About library functions:

You can use the following library functions in your code:

- List.length for obtaining the length of a list e.g. `List.length [1;2;3;4] = 4`
- List.rev for reversing a list e.g. `List.rev [1;2;3;4] = [4;3;2;1]`
- The `@` operator for concatenating two lists e.g. `[1; 2] @ [3; 4] = [1;2;3;4]`.

We have talked about the implementation of these functions in class.

You are not allowed to use any other functions from the OCaml List module (library).

Submission:

Please submit `assignment1.ml` to Canvas.

Problem 1

Write a function

```
pow: int -> int -> int
```

such that `pow x p` returns `x` raised to the power `p`. We assume `p` is non-negative, and we will not test your code for integer overflow cases.

In [ ]:
```
let rec pow x p =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (pow 3 1 = 3)
assert (pow 3 2 = 9)
assert (pow (-3) 3 = -27)
```

Problem 2

Write a function

```
range : int -> int -> int list
```

such that `range num1 num2` returns an ordered list of all integers from `num1` to `num2` inclusive. For example, `range 2 5 = [2;3;4;5]`. Return `[]` if `num2 < num1`.

In [ ]:
```
let rec range num1 num2 =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (range 2 5 = [2;3;4;5])
```

Problem 3

Write a function

```
flatten : 'a list list -> 'a list
```

that flattens a list. For example, `flatten [[1;2];[4;3]] = [1;2;4;3]`.

In [ ]:
```
let rec flatten l =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (flatten ([[1;2];[4;3]]) = [1;2;4;3])
```

Problem 4

Write a function

```
remove_stutter : 'a list -> 'a list
```

that removes stuttering from the original list. For example, `remove_stutter [1;2;2;3;1;1;1;4;4;2;2]= [1;2;3;1;4;2]`.

In [ ]:
```
let rec remove_stutter l =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (remove_stutter [1;2;2;3;1;1;1;4;4;2;2] = [1; 2; 3; 1; 4; 2])
```

Problem 5

Write a function

```
rotate: 'a list -> int -> 'a list
```

such that `rotate l n` rotates the input list `n` places to the right. We assume `0 <= n < length(l)`.

In [ ]:
```
let rotate l n =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (rotate ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"] 2 = ["g"; "h"; "a"; "b"; "c"; "d"; "e"; "f"]);
assert (rotate ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"] 0 = ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"]);
assert (rotate ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"] 7 = ["b"; "c"; "d"; "e"; "f"; "g"; "h"; "a"]);
```

Problem 6

Write a function

```
jump: 'a list -> 'a list -> 'a list
```

such that given two lists, `lst1` and `lst2`, `jump lst1 lst2` returns a list with every odd indexed elements in `lst1`, and every even indexed elements in `lst2`, interwoven together (starting from index 0). Consider 0 as even. If the lists are not the same length, the resulting list should have the length of the shorter of the input lists.

In [ ]:
```
let jump lst1 lst2 =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (jump ["first"; "second"; "third"; "fourth"] ["fifth"; "sixth"; "seventh"; "eighth"] =
          ["fifth"; "second"; "seventh"; "fourth"]);
assert (jump [1; 3; 5; 7] [0; 2; 4; 6; 8] = [0; 3; 4; 7]);
assert (jump ["a"; "b"] ["c"] = ["c"]);
```

Problem 7

Write a function

```
nth: 'a list -> int -> 'a list
```

such that `nth l n` returns every `n`'th element of the list `l`. We assume `1 <= n <= length(l)`.

In [ ]:
```
let nth l n =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (nth [1; 2; 3; 4; 5; 6; 7] 1 = [1; 2; 3; 4; 5; 6; 7]);
assert (nth [1; 2; 3; 4; 5; 6; 7] 2 = [2; 4; 6]);
assert (nth [1; 2; 3; 4; 5; 6; 7] 3 = [3; 6]);
```

Problem 8 (3 questions)

Write an OCaml function

```
digitsOfInt : int -> int list
```

such that `digitsOfInt n` returns `[]` if `n` is less than zero, and returns the list of digits of `n` in the order in which they appear in `n`.

In [ ]:
```
let rec digitsOfInt n =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (digitsOfInt 3124 = [3;1;2;4]);
assert (digitsOfInt 352663 = [3;5;2;6;6;3]);
```

Consider the process of taking a number, adding its digits, then adding the digits of the number derived from it, etc., until the remaining number has only one digit. The number of additions required to obtain a single digit from a number `n` is called the additive persistence of `n`, and the digit obtained is called the digital root of `n`. For example, the sequence obtained from the starting number 9876 is 9876, 30, 3, so 9876 has an additive persistence of 2 and a digital root of 3.

Write two OCaml functions:

```
additivePersistence : int -> int
```

```
digitalRoot : int -> int
```

that take positive integer arguments `n` and return respectively the additive persistence and the digital root of `n`.

In [ ]:
```
let additivePersistence n =
  (* YOUR CODE HERE *)

let digitalRoot n =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (additivePersistence 9876 = 2);
assert (digitalRoot 9876 = 3);
```