Contents

Working through the list of problems here. It's not actually 99 problems, just 85. So I guess it's good that they changed the name.

1 99 OCaml Problems [42/85] [49%]

1.1 Lists [27/28]

1.1.1 DONE 1 Tail of a list

Write a function `last :  'a list -> 'a option` that returns the last element of a list.

```
let rec last lst = match lst with
  | [] -> None
  | x :: [] -> Some x
  | x :: xs -> last xs;;
```

```
val last : 'a list -> 'a option = <fun>
```

Quick test:

```
[last [1;2;3];
 last [1];
 last []]
```

```
- : int option list = [Some 3; Some 1; None]
```

1.1.2 DONE 2 Last two elements of a list

Find the last but one (last and penultimate) elements of a list.

This is very strangely phrased, but at least the title seems clear. Inferring the signature from their example, I'm writing this as a function `last_two:  'a list -> ('a * 'a) option`.

```
let rec last_two lst = match lst with
  | [] -> None
  | x :: [] -> None
  | x :: y :: [] -> Some (x, y)
  | x :: xs -> last_two xs;;
```

```
val last_two : 'a list -> ('a * 'a) option = <fun>
```

Quick tests:

```
[last_two [1;3;2;4;3;2;3];
 last_two [1;3];
 last_two [1];
 last_two []]
```

```
- : (int * int) option list = [Some (2, 3); Some (1, 3); None; None]
```

1.1.3 DONE 3 Nth element of a list

Find the Kth element of a list.

This one seems to need the parentehses around the inner `match` expression. Otherwise, it thinks `m` is of type `'a list`.

```
let rec at n lst = match n with
  | 0 -> (match lst with
          | [] -> None
          | x :: xs -> Some x)
  | m -> (match lst with
          | [] -> None
          | x :: xs -> at (m - 1) xs);;
```

```
val at : int -> 'a list -> 'a option = <fun>
```

Tests:

```
[at 0 [1;2;3;4;5];
 at 1 [1;2;3;4;5];
 at 2 [1;2;3;4;5];
 at 3 [1;2;3;4;5];
 at 4 [1;2;3;4;5];
 at 9 [1;2;3;4;5]]
```

```
- : int option list = [Some 1; Some 2; Some 3; Some 4; Some 5; None]
```

1.1.4 DONE 4 length of a list

Find the number of elements of a list

```
let length lst =
  let rec length_acc i lst = match lst with
    | [] -> i
    | x :: xs -> length_acc (i + 1) xs in
  length_acc 0 lst;;
```

```
val length : 'a list -> int = <fun>
```

```
[length [1;2;3;4;5];
 length [[1;2;3];[4;5]];
 length []]
```

```
- : int list = [5; 2; 0]
```

1.1.5 DONE 5 Reverse a list

Reverse a list

(This isn't tail recursive. Can it be?)

```
let rec rev lst = match lst with
  | [] -> []
  | x :: xs -> (rev xs) @ (x::[]);;
```

```
val rev : 'a list -> 'a list = <fun>
```

```
rev [1;2;5;4;3]
```

```
- : int list = [3; 4; 5; 2; 1]
```

1.1.6 DONE 6 Palindrome

Find out whether a list is a palindrom

```
let rec is_palindrome lst =
  let revlst = rev lst in
  let rec list_equals l1 l2 = match l1 with
    | [] -> (match l2 with
              | [] -> true
              | y :: ys -> false)
    | x :: xs -> (match l2 with
                  | [] -> false
                  | y :: ys -> (match y with
                                | y when y = x -> list_equals xs ys
                                | _ -> false)) in
  list_equals lst revlst;;
```

```
val is_palindrome : 'a list -> bool = <fun>
```

Tests:

```
[is_palindrome [1;2;2;1];
 is_palindrome [1];
 is_palindrome [];
 is_palindrome [1;2;3;4;5;4;3;2;1];
 is_palindrome [1;2;3;4;3]; (* false*)
 is_palindrome [1;2;3]] (* false *)
```

```
- : bool list = [true; true; true; true; false; false]
```

1.1.7 DONE 7 Flatten a list

Flatten a nested list structure

```
(* type definition for nested list *)
type 'a node =
  | One of 'a
  | Many of 'a node list;;


let rec flatten nl = match nl with
  | [] -> []
  | (One x) :: xs -> x :: flatten xs
  | (Many xs) :: xss -> (flatten xs) @ (flatten xss);;


flatten [One "a"; Many [One "b"; Many [One "c" ;One "d"]; One "e"]]
```

```
- : string list = ["a"; "b"; "c"; "d"; "e"]
```

1.1.8 DONE 8 Eliminate duplicates

Eliminate consecutive duplicates of list elements.

```
let rec compress l = match l with
  | [] -> []
  | x :: [] -> x :: []
  | x :: y :: xs -> if (x = y)
                    then compress (y :: xs)
                    else x :: compress (y ::xs)
```

```
val compress : 'a list -> 'a list = <fun>
```

 Test it:

```
compress [1;1;1;1;2;2;2;2;3;3;4;4;5;5;6;5;4]
```

```
- : int list = [1; 2; 3; 4; 5; 6; 5; 4]
```

1.1.9 DONE 9 Pack consecutive duplicates

Pack consecutive duplicates of list elements into sublists

```
let pack l =
  let rec pack_help h l = match h with
    | [] -> (match l with
             | [] -> []
             | x :: xs -> pack_help [x] xs)
    | y :: ys -> (match l with
                  | [] -> [h]
                  | x :: xs -> match x with
                               | x when x = y -> pack_help (x :: h) xs
                               | _ -> h :: (pack_help [x] xs)) in
  pack_help [] l;;
```

```
val pack : 'a list -> 'a list list = <fun>
```

Test

```
pack [1;1;1;2;2;3;3;3;3;3;4;5;6;4]
```

```
- : int list list = [[1; 1; 1]; [2; 2]; [3; 3; 3; 3; 3]; [4]; [5]; [6]; [4]]
```

1.1.10 DONE 10 Run length encoding

Run-length encoding of a list
    Using the previous problem's `pack` function:

```
let encode l =
  let rle x = (List.length x, List.hd x) in
  l |> pack |> List.map rle;;
```

```
val encode : 'a list -> (int * 'a) list = <fun>
```

Test:

```
encode [1;1;1;1;2;3;4;4;4;4;4;4;4;4;3;3;2]
```

```
- : (int * int) list = [(4, 1); (1, 2); (1, 3); (8, 4); (2, 3); (1, 2)]
```

1.1.11 DONE 11 Modified Run-length encoding

Modify the result of the previous problem in such a way that if an element has no duplicates it is simply copied into the result list. Only elements with duplicates are transferred as (N E) lists.
    Since OCaml lists are homogeneous, one needs to define a type to hold both single elements and sub-lists.

```
type 'a rle =
  | One of 'a
  | Many of int * 'a
```

```
type 'a rle = One of 'a | Many of int * 'a
```

(Adding the error to suppress the "incomplete match" warning, but that case should be impossible to reach.)

```
let encode lst =
  let rle_of_packed l = match l with
    | x :: [] -> One x
    | x :: xs -> Many (List.length l, x)
    | [] -> failwith "Error: empty list in packed list" in
  lst |> pack |> List.map rle_of_packed;;
```

```
val encode : 'a list -> 'a rle list = <fun>
```

Test it:

```
encode [1;1;2;2;3;3;3;4;5;5;5;5;5;5;5];;
```

```
- : int rle list =
[Many (2, 1); Many (2, 2); Many (3, 3); One 4; Many (7, 5)]
```

1.1.12   DONE 12 Decode a run-length encoded list

Given a run-length code list generated as specified in the previous problem, construct its uncompressed version.

Note that the base case of the inner match expression is `2` instead of `1`, because `Many (n, x)` can (by construction) only have a value of `n` that's greater than or equal to `2`.

```
let decode lst =
  let rec unpack e = match e with
    | One x -> [x]
    | Many (n,x) -> (match n with
                    | 2 -> x :: x :: []
                    | _ -> x :: unpack (Many (n-1,x))) in
  lst |> List.map unpack |> List.fold_left (@) [];;
```

```
val decode : 'a rle list -> 'a list = <fun>
```

```
decode [Many (2, 1); Many (2, 2); Many (3, 3); One 4; Many (7, 5)]
```

```
- : int list = [1; 1; 2; 2; 3; 3; 3; 4; 5; 5; 5; 5; 5; 5; 5]
```

can this be done without the `fold`? Seems like it might be inefficient (though quick to code).

1.1.13   DONE 13 Run-length encoding of a list (direct solution)

Implement the so-called run-length encoding data compression method directly. I.e. don't explicitly create the sublists containing the duplicates, as in problem "Pack consecutive duplicates of list elements into sublists", but only count them. As in problem "Modified run-length encoding", simplify the result list by replacing the singleton lists (1 X) by X.

```
let encode lst =
  let rec encode_acc ct e lst = match lst with
    | [] -> (match ct with
              | 1 -> [One e]
              | n -> [Many (n,e)])
    | x :: [] when x = e -> [Many (ct + 1, e)]
    | x :: [] -> (match ct with
                  | 1 -> [One e; One x]
                  | n -> [Many (ct, e); One x])
    | x :: xs when x = e -> encode_acc (ct + 1) e xs
    | x :: xs -> (match ct with
                  | 1 -> (One e) :: encode_acc 1 x xs
                  | n -> (Many (n,e)) :: encode_acc 1 x xs) in
  match lst with
  | [] -> []
  | x :: xs -> encode_acc 1 x xs;;
```

```
val encode : 'a list -> 'a rle list = <fun>
```

Test it:

```
encode [1;1;1;1;2;2;3;3;3;3;4;5;6;5;4;4;4;4;5;5;5;5;5;5;5;5;5;0];;
```

```
- : int rle list =
[Many (4, 1); Many (2, 2); Many (4, 3); One 4; One 5; One 6; One 5;
 Many (4, 4); Many (9, 5); One 0]
```

1.1.14 DONE 14 Duplicate the elements of a list

Duplicate the elements of a list

```
let rec duplicate lst = match lst with
  | [] -> []
  | x :: xs -> x :: x :: duplicate xs;;
```

```
val duplicate : 'a list -> 'a list = <fun>
```

```
duplicate ["a";"b";"c";"c";"d"]
```

```
- : string list = ["a"; "a"; "b"; "b"; "c"; "c"; "c"; "c"; "d"; "d"]
```

1.1.15 DONE 15 Replicate the elements of a list a given number of times

Replicate the elements of a list a given number of times

```
let rec replicate lst n =
  let rec repeated n e = match n with
    | 0 -> []
    | n -> e :: repeated (n-1) e in
  lst |> List.map (repeated n) |> List.fold_left (@) [];;
```

```
val replicate : 'a list -> int -> 'a list = <fun>
```

```
replicate [1;2;3;3;4] 4
```

```
- : int list = [1; 1; 1; 1; 2; 2; 2; 2; 3; 3; 3; 3; 3; 3; 3; 3; 4; 4; 4; 4]
```

1.1.16   DONE 16 Drop every N'th element from a list

Drop every N'th element from a list

```
let drop lst n =
  let rec drop_help lst n m = match m with
    | 1 -> (match lst with
            | [] -> []
            | x :: xs -> drop_help xs n n)
    | m -> (match lst with
            | [] -> []
            | x :: xs -> x :: (drop_help xs n (m-1))) in
  drop_help lst n n;;
```

```
val drop : 'a list -> int -> 'a list = <fun>
```

   Test:

```
drop [1;2;3;4;5;6;7;8;9;10] 3
```

```
- : int list = [1; 2; 4; 5; 7; 8; 10]
```

1.1.17   DONE 17 Split a list into two parts; the length of the first part is given

Split a list into two parts; the length of the first part is given
   If the length of the first part is longer than the entire list, then the first part is the list and the second
part is empty.

```
let split lst n =
  let rec split_help lst partial n = match n with
    | 0 -> [List.rev partial; lst]
    | n -> (match lst with
            | [] -> [List.rev partial;lst]
            | x :: xs -> split_help xs (x :: partial) (n-1)) in
  split_help lst [] n;;
```

```
val split : 'a list -> int -> 'a list list = <fun>
```

   Tests:

```
[split [1;2;3;4;5;6;7] 0;
 split [1;2;3;4;5;6;7] 1;
 split [1;2;3;4;5;6;7] 4;
 split [1;2;3;4;5;6;7] 12]
```

```
- : int list list list =
[[[]; [1; 2; 3; 4; 5; 6; 7]]; [[1]; [2; 3; 4; 5; 6; 7]];
 [[1; 2; 3; 4]; [5; 6; 7]]; [[1; 2; 3; 4; 5; 6; 7]; []]]
```

1.1.18 DONE 18 Extract a slice from a list

Given two indices, `i` and `k`, the slice is the list containing the elements between the `ith` and `kth` element of the original list (both limits included). Start counting the elements with `0` (this is the way the List module numbers elements).

```
let rec slice lst i j = match i with
  | 0 -> (match j with
         | 0 -> (match lst with
                | [] -> []
                | x :: xs -> [x])
         | j when j > 0 -> (match lst with
                           | [] -> []
                           | x :: xs -> x :: (slice xs 0 (j-1)))
         | j -> [])
  | i -> (match lst with
         | [] -> []
         | x :: xs -> slice xs (i-1) (j-1));;
```

```
val slice : 'a list -> int -> int -> 'a list = <fun>
```

Test:

```
slice [1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17] 5 7
```

```
- : int list = [6; 7; 8]
```

1.1.19 DONE 19 Rotate a list N places to the left

Rotate a list N places to the left
    Can be a little clever here with modular arithmetic to avoid wasting a bunch of time:

```
let rotate lst n =
  let l = List.length lst in
  let m = if (n mod l >= 0) then (n mod l) else ((n mod l) + l) in
  let rec rotate_help lst part n = match n with
    | 0 -> lst @ part
    | n -> (match lst with
           | [] -> part
           | x :: xs -> rotate_help xs (part @ [x]) (n-1)) in
  rotate_help lst [] m;;
```

```
val rotate : 'a list -> int -> 'a list = <fun>
```

```
[rotate [1;2;3;4;5;6;7] (-8);
 rotate [1;2;3;4;5;6;7] (1000);
 rotate [1] (100000);
 rotate [1;2;3;4;5;6;7] (-12367)]
```

```
- : int list list =
[[7; 1; 2; 3; 4; 5; 6]; [7; 1; 2; 3; 4; 5; 6]; [1]; [3; 4; 5; 6; 7; 1; 2]]
```

1.1.20 DONE 20 Remove the K'th element from a list

Remove the K'th element from a list

The first element of the list is numbered 0, the second 1,...

```
let remove_at k lst =
  let rec remove_at_help k lst partial = match k with
    | 0 -> (match lst with
            | [] -> partial
            | x :: xs -> partial @ xs)
    | k -> (match lst with
            | [] -> partial
            | x :: xs -> remove_at_help (k-1) xs (partial @ [x]))  in
  remove_at_help k lst [];;
```

```
val remove_at : int -> 'a list -> 'a list = <fun>
```

Test

```
remove_at 3 [1;2;3;4;5;6;7];;
```

```
- : int list = [1; 2; 3; 5; 6; 7]
```

1.1.21 DONE 21 Insert element into a list at a given position

Start counting list elements with 0. If the position is larger or equal to the length of the list, insert the element at the end. (The behavior is unspecified if the position is negative.)

```
let rec insert_at e i lst =
  match i with
  | j when j <= 0 -> e :: lst
  | i -> (match lst with
          | [] -> [e]
          | x :: xs -> x :: (insert_at e (i-1) xs));;
```

```
val insert_at : 'a -> int -> 'a list -> 'a list = <fun>
```

```
insert_at 2 4 [1;1;1;1;1;1;1;1;1]
```

```
- : int list = [1; 1; 1; 1; 2; 1; 1; 1; 1; 1]
```

(not tail recursive. can be re-written to be so, but I can only see a way that might overuse the @ operator)

1.1.22   DONE 22 Create a list containing all integers within a given range

Create a list containing all integers within a given range. If first argument is greater than second, produce a list in decreasing order

```
let rec range i j =
  let k = j - i in
  match k with
  | k when k < 0 -> i :: (range (i-1) j)
  | k when k = 0 -> [i]
  | k -> i :: range (i+1) j;;
```

```
val range : int -> int -> int list = <fun>
```

```
[range (-10) (-2);
 range 1 11;
 range 4 4;
 range 10 0]
```

```
- : int list list =
[[-10; -9; -8; -7; -6; -5; -4; -3; -2]; [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11];
 [4]; [10; 9; 8; 7; 6; 5; 4; 3; 2; 1; 0]]
```

1.1.23   DONE 23 Extract a given number of randomly selected elements from a list

The selected items shall be returned in a list. We use the Random module but do not initialize it with `Random.self_init` for reproducibility.

(I'm assuming this means the elements should be distinct? as in, a random subset of the specified size?)

If the list has length `n` and you're picking `k` elements, then there are `n` choose `k` subsets. And `n-1` choose `k-1` of them will contain the first element. So with probability $\frac{k}{n}$, pick the first element, and recursively choose `k-1` elements in the tail of the list. But with probability $1 - \frac{k}{n}$, don't pick the first element, and instead pick `k` elements from the tail of the list.

```
let rec rand_select lst k =
  let n = List.length lst in
  match k with
  | k when k > n -> []
  | k when k = n -> lst
  | k -> let i = Random.int n in
         match lst with
         | [] -> []
         | x :: xs -> if i + 1 <= k
                      then (x :: rand_select xs (k-1))
                      else (rand_select xs k);;
```

```
val rand_select : 'a list -> int -> 'a list = <fun>
```

```
[rand_select [1;2;3;4;5;6;7] 3;
 rand_select [1;2;3;4;5;6;7] 3;
 rand_select [1;2;3;4;5;6;7] 3;
 rand_select [1;2;3;4;5;6;7] 2;
 rand_select [1;2;3;4;5;6;7] 2;
 rand_select [1;2;3;4;5;6;7] 2;
 rand_select [1;2;3;4;5;6;7] 2]
```

```
- : int list list =
[[3; 5; 7]; [5; 6; 7]; [1; 2; 7]; [3; 7]; [3; 6]; [2; 5]; [2; 7]]
```

Looks pretty random to me. Should probably do actual statistics to be sure, but I trust the math.

1.1.24   DONE 24 Lotto: Draw N different random numbers from the set 1..M

Draw N different random numbers from the set $\{1...M\}$. The selected numbers shall be returned in a list. There's really not much to it if you use the previous problem.

```
let lotto_select n m = rand_select (range 1 m) n;;
```

```
val lotto_select : int -> int -> int list = <fun>
```

```
lotto_select 5 50
```

```
- : int list = [4; 22; 29; 32; 46]
```

1.1.25   DONE 25 Generate a random permutation of the elements of a list

Generate a random permutation of the elements of a list
     (this can probably be done more efficiently. Using my `remove_at` from earlier might be bad)

```
let rec permutation lst = match lst with
  | [] -> []
  | _ -> let n = List.length lst in
         let i = Random.int n in
         let h = List.nth lst i in
         h :: permutation (remove_at i lst);;
```

```
val permutation : 'a list -> 'a list = <fun>
```

```
permutation (range 1 100)
```

```
- : int list =
[26; 72; 57; 53; 22; 25; 76; 69; 14; 65; 28; 21; 95; 37; 77; 51; 44; 35; 33;
 20; 38; 46; 84; 32; 82; 92; 98; 96; 8; 31; 18; 24; 89; 60; 74; 67; 73; 80;
 9; 81; 56; 39; 87; 94; 100; 75; 88; 12; 66; 59; 29; 45; 16; 2; 91; 1; 93; 7;
 27; 11; 70; 30; 79; 13; 19; 83; 64; 90; 6; 52; 86; 40; 36; 99; 58; 3; 97;
 34; 10; 50; 4; 63; 71; 55; 42; 48; 85; 49; 5; 78; 62; 17; 61; 41; 54; 15;
 68; 47; 43; 23]
```

1.1.26   DONE 26 Generate the combinations of K distinct objects chosen from the N elements of a list

Generate the combinations of K distinct objects chosen from the N elements of a list.

In how many ways can a committee of 3 be chosen from a group of 12 people? We all know that there are 12 choose 3 = 220 possibilities. For pure mathematicians, this result may be great. But we want to really generate all the possibilities in a list.

```
let rec extract k lst = match k with
  | k when k < 0 -> []
  | 0 -> [[]]
  | k -> (let n = List.length lst in
          match n with
          | n when n < k -> []
          | n when n = k -> [lst]
          | n -> (match lst with
                  | [] -> []
                  | x :: xs ->
                    (List.map (fun s -> x :: s) (extract (k-1) xs))
                    @ (extract k xs)));;
```

```
val extract : int -> 'a list -> 'a list list = <fun>
```

Tests in separate blocks here, for readability
There are no subsets with size −1.

```
extract (-1) [1;2;3;4;5;6]
```

```
- : int list list = []
```

But there's exactly one subset with size 0 (the empty set).

```
extract 0 [1;2;3;4;5;6]
```

```
- : int list list = [[]]
```

There are six subsets of size 1.

```
extract 1 [1;2;3;4;5;6]
```

```
- : int list list = [[1]; [2]; [3]; [4]; [5]; [6]]
```

And $\binom{6}{2} = 15$ subsets of size 2.

```
extract 2 [1;2;3;4;5;6]
```

```
- : int list list =
[[1; 2]; [1; 3]; [1; 4]; [1; 5]; [1; 6]; [2; 3]; [2; 4]; [2; 5]; [2; 6];
 [3; 4]; [3; 5]; [3; 6]; [4; 5]; [4; 6]; [5; 6]]
```

There's only one subset of size 6.

```
extract 6 [1;2;3;4;5;6]
```

```
- : int list list = [[1; 2; 3; 4; 5; 6]]
```

1.1.27 TODO 27 - Group the elements of a list into disjoint subsets

Group the elements of a set into disjoint subsets

- In how many ways can a group of 9 people work in 3 disjoint subgroups of 2, 3 and 4 persons? Write a function that generates all the possibilities and returns them in a list.

- Generalize the above function in a way that we can specify a list of group sizes and the function will return a list of groups.

1.1.28 DONE 28 Sorting a list of lists according to length of sublists

Sorting a list of lists according to length of sublists.

- We suppose that a list contains elements that are lists themselves. The objective is to sort the elements of this list according to their length. E.g. short lists first, longer lists later, or vice versa.

- Again, we suppose that a list contains elements that are lists themselves. But this time the objective is to sort the elements of this list according to their length frequency; i.e., in the default, where sorting is done ascendingly, lists with rare lengths are placed first, others with a more frequent length come later.

```
let length_sort lst =
  let ( <<< ) l1 l2 = List.length l1 < List.length l2 in
  let rec qs lst comparison = match lst with
    | [] -> []
    | x :: xs -> (let in_left l = l <<< x in
                  let (left, right) = List.partition in_left xs in
                  (qs left (<<<)) @ [x] @ (qs right (<<<))) in
  qs lst (<<<);;
```

```
val length_sort : 'a list list -> 'a list list = <fun>
```

```
length_sort [[1;2;3];[4];[5;6];[7;7];[]]
```

```
- : int list list = [[]; [4]; [5; 6]; [7; 7]; [1; 2; 3]]
```

1.2 Arithmetic [5/11]

1.2.1 TODO 29 Primality test

Determine whether a given integer is prime
    Ordinary naive seive:

```ocaml
let is_prime_seive n =
  if n < 2
  then false
  else (let bound = n
                  |> float_of_int
                  |> Float.sqrt
                  |> Float.floor
                  |> int_of_float in
        let rec range a b =
          let s = b - a in
          match s with
          | s when s < 0 -> []
          | 0 -> [a]
          | s -> a :: range (a+1) b in
        let candidates = range 2 bound in
        (* seive lst m = true if a number in lst,
         is a divisor of m, false otherwise. *)
        let rec seive lst m = match lst with
          | [] -> (false)
          | p :: qs -> (if m mod p = 0
                        then true
                        else seive qs m) in
        not (seive candidates n));;

let rec range a b =
  let s = b - a in
  match s with
  | s when s < 0 -> []
  | 1 -> [a]
  | s -> a :: range (a+1) b;;

List.filter is_prime_seive (range 1 100000)
```

```
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71;
 73; 79; 83; 89; 97; 101; 103; 107; 109; 113; 127; 131; 137; 139; 149; 151;
 157; 163; 167; 173; 179; 181; 191; 193; 197; 199; 211; 223; 227; 229; 233;
 239; 241; 251; 257; 263; 269; 271; 277; 281; 283; 293; 307; 311; 313; 317;
 331; 337; 347; 349; 353; 359; 367; 373; 379; 383; 389; 397; 401; 409; 419;
 421; 431; 433; 439; 443; 449; 457; 461; 463; 467; 479; 487; 491; 499; 503;
 509; 521; 523; 541; 547; 557; 563; 569; 571; 577; 587; 593; 599; 601; 607;
 613; 617; 619; 631; 641; 643; 647; 653; 659; 661; 673; 677; 683; 691; 701;
 709; 719; 727; 733; 739; 743; 751; 757; 761; 769; 773; 787; 797; 809; 811;
 821; 823; 827; 829; 839; 853; 857; 859; 863; 877; 881; 883; 887; 907; 911;
 919; 929; 937; 941; 947; 953; 967; 971; 977; 983; 991; 997; 1009; 1013;
 1019; 1021; 1031; 1033; 1039; 1049; 1051; 1061; 1063; 1069; 1087; 1091;
 1093; 1097; 1103; 1109; 1117; 1123; 1129; 1151; 1153; 1163; 1171; 1181;
 1187; 1193; 1201; 1213; 1217; 1223; 1229; 1231; 1237; 1249; 1259; 1277;
 1279; 1283; 1289; 1291; 1297; 1301; 1303; 1307; 1319; 1321; 1327; 1361;
 1367; 1373; 1381; 1399; 1409; 1423; 1427; 1429; 1433; 1439; 1447; 1451;
 1453; 1459; 1471; 1481; 1483; 1487; 1489; 1493; 1499; 1511; 1523; 1531;
 1543; 1549; 1553; 1559; 1567; 1571; 1579; 1583; 1597; 1601; 1607; 1609;
 1613; 1619; 1621; 1627; 1637; 1657; 1663; 1667; 1669; 1693; 1697; 1699;
 1709; 1721; 1723; 1733; 1741; 1747; 1753; 1759; 1777; 1783; 1787; 1789;
 1801; 1811; 1823; 1831; 1847; 1861; 1867; 1871; 1873; 1877; 1879; 1889;
 1901; 1907; 1913; 1931; 1933; 1949; 1951; 1973; 1979; ...]
```

Miller-Rabin:

```
let is_prime n = match n with
  | n when n < 2 -> false
  | 2 -> true
  | n when n mod 2 = 0 -> false
  | n -> (let witnesses = [2;3;5;7;11;13;17;19;23;29;31;37] in
          let rec twos_exponent m =
            match m with
            | m when m mod 2 = 1 -> 0
            | m -> 1 + twos_exponent (m / 2) in
          let rec power a b = match b with
            | 0 -> 1
            | b -> a * pow a (b-1) in
          let rec powermod a b n = match b with
            | 0 -> 1
            | b -> (a * powermod a (b-1) n) mod n in
          let fermat_test a d n =
            powermod a d n = 1 in
          let rec root_test a r s n = () in
          let s = twos_exponent (n-1) in
          let d = (n-1) / (pow 2 s) in
          (
          (* miller-rabin goes here*)
          )
         );;
is_prime 1001
```

```
Line 1, characters 0-8:
1 | is_prime 1001;;
    ^^^^^^^^
Error: Unbound value is_prime
Hint: Did you mean coprime?
```

1.2.2 DONE 30 - Determine the greatest common divisor of two positive integer numbers

Determine the greatest common divisor of two positive integer numbers.
Use Euclid's algorithm.

```
let rec gcd a b =
  if (a < b)
  then (gcd b a)
  else let q = a / b in
       let r = a - q*b in
       match r with
       | 0 -> b
       | r -> gcd b r;;

gcd (324*17*11*13) (324*2*5*101);;
```

```
- : int = 324
```

1.2.3 DONE 31 - Determine whether two positive integer numbers are coprime

Determine whether two positive integer numbers are coprime.

Two numbers are coprime if their greatest common divisor equals 1.
(seems trivial)

```
let rec coprime a b = gcd a b = 1;;
```

```
val coprime : int -> int -> bool = <fun>
```

1.2.4 DONE 32 - Calculate Euler's totient function $\phi(m)$

Euler's so-called totient function $\varphi(m)$ is defined as the number of positive integers $1 \leqslant r \leqslant m$ that are coprime to m. We let $\varphi(1) = 1$

Find out what the value of $\varphi(m)$ is if m is a prime number. Euler's totient function plays an important role in one of the most widely used public key cryptography methods (RSA). In this exercise you should use the most primitive method to calculate this function (there are smarter ways that we shall discuss later).

(doing it the naive way:)

```
let phi m = match m with
  | 1 -> 1
  | m -> (let range a b =
            let s = b - a in
            match s with
            | s when s < 0 -> []
            | 0 -> [a]
            | s -> a :: range (a+1) b in
         let rec count_coprimes acc lst n =
           match lst with
           | [] -> acc
           | d :: ds -> if (gcd n d = 1)
                        then (count_coprimes (acc+1) ds n)
                        else (count_coprimes acc ds n) in
         count_coprimes 0 (range 1 m) m);;

phi 12321
```

```
- : int = 7992
```

1.2.5 TODO 33 - Determine the prime factors of a given positive integer

BUGGY - check n = 25, it's failing there for some reason

Construct a flat list containing the prime factors in ascending order.

```
let rec factors n =
  if is_prime_seive n
  then [n]
  else
    let bound = n
                |> float_of_int
                |> Float.sqrt
                |> Float.floor
                |> int_of_float in
    let range a b =
      let s = b - a in
      match s with
      | s when s < 0 -> []
      | 0 -> [a]
      | s -> a :: range(a+1) b in
    let potential_divisors = range 2 bound
                             |> List.filter is_prime_seive in
    let rec smallest_prime_factor lst m = match lst with
      | [] -> 1 (* impossible branch since n is composite *)
      | d :: ds -> if m mod d = 0 then d else smallest_prime_factor ds m in
    let p = smallest_prime_factor potential_divisors n in
    p :: factors (n/p);;

factors (324*72*17*11)
```

```
- : int list = [2; 2; 2; 2; 2; 3; 3; 3; 3; 3; 3; 11; 17]
```

1.2.6   DONE 34 - Determine the prime factors of a given positive integer (2)

Construct a list containing the prime factors and their multiplicity. Hint: The problem is similar to problem 13

    doing it the naive way for now:

```
let factors_with_multiplicity n =
  let rec compress count p lst = match lst with
    | [] -> [(p,count)]
    | x :: xs when x = p -> compress (count+1) p xs
    | x :: xs ->(p,count) :: compress 1 x xs in
  match factors n with
  | [] -> []
  | [p] -> [(p,1)]
  | p :: ps -> compress 1 p ps;;

factors_with_multiplicity (324*72*17*11*37)
```

```
- : (int * int) list = [(2, 5); (3, 6); (11, 1); (17, 1); (37, 1)]
```

1.2.7 TODO 35 Calculate Euler's totient function (improved)

```
let eulerphi m =
  let facts = factors_with_multiplicity m in
  let rec exp a b = match b with
    | 0 -> 1
    | b -> a * (exp a (b-1)) in
  let rec phi_list_product lst = match lst with
    | [] -> 1
    | (p,e) :: tail -> (p-1) * (exp p (e-1)) * phi_list_product tail in
  phi_list_product facts;;

(* check that it agrees w/ the previous *)
  factors 50
```

```
Stack overflow during evaluation (looping recursion?).
```

1.2.8 TODO 36 Compare the two methods of calculating Euler's totient function

1.2.9 DONE 37 A list of prime numbers

Given a range of integers by its lower and upper limit, construct a list of all prime numbers in that range.

```
let all_primes a b =
  let range a b =
    let s = b - a in
    match s with
    | s when s < 0 -> []
    | 0 -> [a]
    | s -> a :: range (a+1) b in
  range a b
  |> List.filter is_prime_seive;;

(* could be improved *)

List.length (all_primes 2 7920)
```

```
- : int = 1000
```

1.2.10 TODO 38 Goldbach's conjecture

1.2.11 TODO 39 A list of Goldbach compositions

1.3 Logic and Codes [1/4]

1.3.1 TODO 40 Truth tables for logical expressions (2 variables)

1.3.2 TODO 41 Truth tables for logical expressions

1.3.3 DONE 42 Gray code

An n-bit Gray code is a sequence of n-bit strings constructed according to certain rules. For example,

n = 1: C(1) = ['0', '1']. n = 2: C(2) = ['00', '01', '11', '10']. n = 3: C(3) = ['000', '001', '011', '010', '110', '111', '101', '100'].

Find out the construction rules and write a function with the following specification: gray n returns the n-bit Gray code.

So vague...

```
let rec gray n = match n with
  | 0 -> [""]
  | n -> (List.map ((^) "0") (gray (n-1))) @
           (List.map ((^) "1") (gray (n-1) |> List.rev));;

gray 3
```

```
- : string list = ["000"; "001"; "011"; "010"; "110"; "111"; "101"; "100"]
```

1.3.4 TODO 43 Huffman code

1.4 Trees [9/17]

1.4.1 DONE 44 Completely balanced binary trees

A binary tree is either empty or it is composed of a root element and two successors, which are binary trees themselves.

In OCaml, one can define a new type `binary_tree` that carries an arbitrary value of type `'a` (thus is polymorphic) at each node.

```
type 'a binary_tree =
  | Empty
  | Node of 'a * 'a binary_tree * 'a binary_tree;;
type 'a binary_tree = Empty | Node of 'a * 'a binary_tree * 'a binary_tree
```

```
type 'a binary_tree = Empty | Node of 'a * 'a binary_tree * 'a binary_tree
```

An example of tree carrying `char` data is:

```
let example_tree =
  Node ('a', Node ('b', Node ('d', Empty, Empty), Node ('e', Empty, Empty)),
        Node ('c', Empty, Node ('f', Node ('g', Empty, Empty), Empty)));;
```

```
val example_tree : char binary_tree =
  Node ('a', Node ('b', Node ('d', Empty, Empty), Node ('e', Empty, Empty)),
   Node ('c', Empty, Node ('f', Node ('g', Empty, Empty), Empty)))
```

In OCaml, the strict type discipline guarantees that, if you get a value of type `binary_tree`, then it must have been created with the two constructors `Empty` and `Node`.

In a completely balanced binary tree, the following property holds for every node: The number of nodes in its left subtree and the number of nodes in its right subtree are almost equal, which means their difference is not greater than one.

Write a function `cbal_tree` to construct completely balanced binary trees for a given number of nodes. The function should generate all solutions via backtracking. Put the letter `'x'` as information into all nodes of the tree.

```
let rec cbal_tree n =
  let rec outer f lst1 lst2 = match lst1 with
  | [] -> []
  | x :: xs -> (List.map (fun y -> f x y) lst2)
             @ outer f xs lst2 in
  let join l r = Node ('x', l, r) in
  let all_joins llist rlist = (outer join llist rlist) in
  match n with
  | 0 -> [Empty]
  | 1 -> [Node('x', Empty, Empty)]
  | n when n mod 2 = 1 -> (let m = (n - 1)/2 in
                          let subtrees = cbal_tree m in
                          all_joins subtrees subtrees)
  | n -> (let a = (n-2)/2 in
         let b = a + 1 in
         let asubtrees = cbal_tree a in
         let bsubtrees = cbal_tree b in
         (all_joins asubtrees bsubtrees)
         @ (all_joins bsubtrees asubtrees));;

[0;1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20;21;22;23;24;25]
|> List.map cbal_tree
|> List.map List.length
```

```
- : int list =
[1; 1; 2; 1; 4; 4; 4; 1; 8; 16; 32; 16; 32; 16; 8; 1; 16; 64; 256; 256; 1024;
 1024; 1024; 256; 1024; 1024]
```

results agree with https://oeis.org/A110316

1.4.2 DONE 45 Symmetric binary trees

Let us call a binary tree symmetric if you can draw a vertical line through the root node and then the right subtree is the mirror image of the left subtree. Write a function `is_symmetric` to check whether a given binary tree is symmetric.

Hint: Write a function `is_mirror` first to check whether one tree is the mirror image of another. We are only interested in the structure, not in the contents of the nodes.

```
let is_symmetric t =
  let rec is_mirror t1 t2 = match t1 with
    | Empty -> (match t2 with
                | Empty -> true
                | _ -> false)
    | Node (x, l1, r1) -> (match t2 with
                           | Empty -> false
                           | Node(y, l2, r2) -> (is_mirror l1 r2)
                                                && (is_mirror l2 r1)) in
  match t with
  | Empty -> true
  | Node (x, l, r) -> is_mirror l r;;

List.map is_symmetric (cbal_tree 9);;
```

```
- : bool list =
[false; false; false; true; false; false; true; false; false; true; false;
 false; true; false; false; false]
```

1.4.3   DONE 46 Binary search trees

Construct a binary search tree from a list of integer numbers.

```
let construct lst =
  let rec insert t e = match t with
    | Empty -> Node(e, Empty, Empty)
    | Node (x, left, right) when e <= x -> Node(x, insert left e, right)
    | Node (x, left, right) -> Node(x, left, insert right e) in
  let rec insert_list t lst = match lst with
    | [] -> t
    | e :: es -> insert_list (insert t e) es in
  insert_list Empty lst;;

construct [3;2;5;7;1]
```

```
- : int binary_tree =
Node (3, Node (2, Node (1, Empty, Empty), Empty),
 Node (5, Empty, Node (7, Empty, Empty)))
```

Then use this function to test the solution of the previous problem.

```
is_symmetric (construct [5; 3; 18; 1; 4; 12; 21]);;
```

```
- : bool = true
```

```
not (is_symmetric (construct [3; 2; 5; 7; 4]));;
```

```
- : bool = true
```

1.4.4 DONE 47 Generate-and-test paradigm

Apply the generate-and-test paradigm to construct all symmetric, completely balanced binary trees with a given number of nodes.

```
let sym_cbal_tree n =
  n
  |> cbal_tree
  |> List.filter is_symmetric;;

sym_cbal_tree 5;;

List.length (sym_cbal_tree 57);;
```

```
- : int = 256
```

For `Node (x, left, right)` to be symmetric, `left` and `right` need to have the same number of nodes. So there will be no symmetric trees with an even number of nodes.

```
let rec range a b = match a with
  | a when a < b -> a :: (range (a+1) b)
  | a when a = b -> [b]
  | _ -> [] in
    (range 1 10)
    |> List.map (fun n -> 2*n)
    |> List.map sym_cbal_tree
    |> List.map List.length;;
```

```
- : int list = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0]
```

But for odd numbers?

```
let rec range a b = match a with
  | a when a < b -> a :: (range (a+1) b)
  | a when a = b -> [b]
  | _ -> [] in
    (range 0 24)
    |> List.map (fun n -> 2*n + 1)
    |> List.map (fun m -> (m, sym_cbal_tree m))
    |> List.map (fun (a,b) -> (a,List.length b));;
```

```
- : (int * int) list =
[(1, 1); (3, 1); (5, 2); (7, 1); (9, 4); (11, 4); (13, 4); (15, 1); (17, 8);
 (19, 16); (21, 32); (23, 16); (25, 32); (27, 16); (29, 8); (31, 1);
 (33, 16); (35, 64); (37, 256); (39, 256); (41, 1024); (43, 1024);
 (45, 1024); (47, 256); (49, 1024)]
```

My guess is that the number of symmetric completely balanced trees with 2n+1 nodes will be the number of completely balanced trees with n nodes, since to be symmetric and completely balanced, it needs to be of the form `Node(x, left, right)` where `left` is a completely balanced tree with `n` nodes. But this completely determined `right`.

```
let rec range a b = match a with
  | a when a < b -> a :: (range (a+1) b)
  | a when a = b -> [b]
  | _ -> [] in
    (range 0 24)
    |> List.map (fun n -> (n, 2*n+1))
    |> List.map (fun (a, b) -> (a |> cbal_tree |> List.length, b |> sym_cbal_tree |>
    ↪  List.length));;
```

```
- : (int * int) list =
[(1, 1); (1, 1); (2, 2); (1, 1); (4, 4); (4, 4); (4, 4); (1, 1); (8, 8);
 (16, 16); (32, 32); (16, 16); (32, 32); (16, 16); (8, 8); (1, 1); (16, 16);
 (64, 64); (256, 256); (256, 256); (1024, 1024); (1024, 1024); (1024, 1024);
 (256, 256); (1024, 1024)]
```

Seems right.

1.4.5 DONE 48 Construct height-balanced binary trees

In a height-balanced binary tree, the following property holds for every node: The height of its left subtree and the height of its right subtree are almost equal, which means their difference is not greater than one.

Write a function `hbal_tree` to construct height-balanced binary trees for a given height. The function should generate all solutions via backtracking. Put the letter `'x'` as information into all nodes of the tree.

```
let rec hbal_tree h =
  let rec outer f lst1 lst2 = match lst1 with
    | [] -> []
    | x :: xs -> (List.map (fun y -> f x y) lst2)
                 @ outer f xs lst2 in
  let join l r = Node ('x', l, r) in
  let all_joins llist rlist = (outer join llist rlist) in
  match h with
  | 0 -> [Empty]
  | 1 -> [Node('x', Empty, Empty)]
  | h -> (let one_shorter_trees = hbal_tree (h-1) in
          let two_shorter_trees = hbal_tree (h-2) in
          (all_joins one_shorter_trees one_shorter_trees)
          @ (all_joins one_shorter_trees two_shorter_trees)
          @ (all_joins two_shorter_trees one_shorter_trees));;

List.length (hbal_tree 3)
```

```
- : int = 15
```

1.4.6 TODO 49 Construct height-balanced binary trees with a given number of nodes

Consider a height-balanced binary tree of height `h`. What is the maximum number of nodes it can contain?

The answer is definitely $2^h - 1$ (just fill the tree). but confirm this by exhaustive search for small h values

```
let max_nodes h =
  let rec node_count t = match t with
    | Empty -> 0
    | Node(x, left, right) -> 1 + (node_count left) + (node_count right) in
  let rec maximum r lst = match lst with
    | [] -> r
    | x :: xs -> if (x > r)
                 then (maximum x xs)
                 else (maximum r xs) in
  h
  |> hbal_tree
  |> List.map node_count
  |> maximum 0;;

List.map max_nodes [1;2;3;4;5]
```

```
- : int list = [1; 3; 7; 15; 31]
```

Seems right. But a better way would be:

```
  let max_nodes h =
  let rec pow a b =
    match b with
    | 0 -> 1
    | b -> a * (pow a (b-1)) in
  (pow 2 h) - 1;;

List.map max_nodes [0;1;2;3;4;5]
```

```
- : int list = [0; 1; 3; 7; 15; 31]
```

(could improve this further with better exponentiation, or even with bit shifting)

What about the minimum number of nodes? Brute force first, to help make a conjecture:

```
let min_nodes h =
  let rec node_count t = match t with
    | Empty -> 0
    | Node(x, left, right) -> 1 + (node_count left) + (node_count right) in
  let rec minimum_acc r lst = match lst with
    | [] -> r
    | x :: xs -> if (x < r)
                 then (minimum_acc x xs)
                 else (minimum_acc r xs) in
  let max = max_nodes h in
  h
  |> hbal_tree
  |> List.map node_count
  |> minimum_acc max;;

List.map min_nodes [0;1;2;3;4;5]
```

```
- : int list = [0; 1; 2; 4; 7; 12]
```

My guess is that min_nodes h is 1 + (min_nodes (h-1)) + (min_nodes (h-2)), with initial terms min_nodes 0 = 0 and min_nodes 1 = 1. Makes sense if you think about trying to construct such a tree of height h using as few nodes as possible: You'd (arbitrarily) want the left tree to have height h-1 and the right to have height h-2, and each of them should have as few nodes as possible. There's some combinatorial details to check though, but here's a faster function:

```
let min_nodes h =
let rec min_nodes_help a b h =
  match h with
  | 0 -> a
  | 1 -> b
  | h -> min_nodes_help (b) (a + b + 1) (h-1) in
min_nodes_help 0 1 h;;

List.map min_nodes [0;1;2;3;4;5;6;7;8;9;10]
```

```
- : int list = [0; 1; 2; 4; 7; 12; 20; 33; 54; 88; 143]
```

Now, just need a way to generate all height-balanced trees with a fixed number of nodes.

1.4.7 DONE 50 Collect the leaves of a binary tree in a list

A leaf is a node with no successors Write a function `leaves` to collect them in a list.

```
  let rec leaves t = match t with
    | Empty -> []
    | Node (x, Empty, Empty) -> [x]
    | Node (x,l,r) -> (leaves l) @ (leaves r);;

  let t = Node ('0',
             Node ('1',
                  Node ('6',
                        Empty,
                        Empty),
                  Node ('3',
                     Node ('7',
                           Empty,
                           Empty),
                     Empty)),
             Node ('2',
               Node ('4',
                    Node ('8',
                          Empty,
                          Empty),
                    Node ('5',
                       Node ('9',
                             Empty,
                             Empty),
                       Empty)),
               Empty));;
leaves t;;
```

```
- : char list = ['6'; '7'; '8'; '9']
```

1.4.8   DONE 51 Count the leaves of a binary tree

A leaf is a node with no successors. Write a function `count_leaves` to count them.

```
let rec count_leaves t = match t with
  | Empty -> 0
  | Node(x,Empty,Empty) -> 1
  | Node(x,left,right) -> (count_leaves right)
                          + (count_leaves left);;

count_leaves Empty;;
count_leaves (Node('x',Node('y',Empty,Empty),Empty));;
count_leaves t
```

```
- : int = 4
```

1.4.9   DONE 52 Collect the nodes at a given level in a list

A node of a binary tree is at level N if the path from the root to the node has length N-1. The root node is at level 1. Write a function `at_level t l` to collect all nodes of the tree `t` at level `l` in a list.

```
let rec at_level t l = match l with
  | l when l < 1 -> []
  | 1 -> (match t with
         | Empty -> []
         | Node (x, l, r) -> [x])
  | l -> (match t with
         | Empty -> []
         | Node (x, left, right) -> (at_level left (l-1)) @
                                    (at_level right (l-1)));;

at_level t 0;;
at_level t 1;;
at_level t 2;;
at_level t 3;;
at_level t 4;;
at_level t 5;;
at_level t 6;;
```

```
- : char list = []
```

1.4.10 DONE 53 Collect the internal nodes of a binary tree in a list

An internal node of a binary tree has either one or two non-empty successors. Write a function `internals` to collect them in a list.

```
let rec internals t = match t with
  | Empty -> []
  | Node (x, Empty, Empty) -> []
  | Node (x, left, right) -> [x]
                             @ (internals left)
                             @ (internals right);;

internals t
```

```
- : char list = ['0'; '1'; '3'; '2'; '4'; '5']
```