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 K^{th} 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 here 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;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).

(This code is ugly, can it be rewritten to maybe look a little nicer? Maybe start with a match on `lst` as well?)

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

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 = [6; 10; 33; 43; 48]
```

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

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 [11/11]

1.2.1 DONE 29 Primality test

Determine whether a given integer is prime

For starters, here's a naive seive:

```
let is_prime_seive n =
  if n < 2
  then false
  else if n = 2 then true
  else (let rec range a b = if a = b
                            then [a]
                            else a :: range (a+1) b in
        let bound = float_of_int n
                    |> Float.sqrt
                    |> Float.ceil
                    |> int_of_float in
        let candidates = range 2 bound in
        let rec seive lst m =
          let rec filter p ns = match ns with
            | [] -> []
            | m :: ms -> if m mod p = 0
                         then filter p ms
                         else m :: filter p ms in
          match lst with
          | [] -> (false)
          | p :: ms -> (if m mod p = 0
                        then true
                        else seive (filter p ms) m) in
        not (seive candidates n));;
```

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

Find list of all primes up to 1000. Check for correctness with Mathematica.

```
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;;
```

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

```
List.filter is_prime_seive (range 1 1000)
```

```
- : 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]
```

Similarly, can count the primes up to a fixed bound, and check whether it agrees with Mathematica's 'PrimePi' function, which it does.

```
range 1 100000
|> List.filter is_prime_seive
|> List.length
```

```
- : int = 9592
```

And we can check its output on large prime (and composite) numbers for which we already know the answers. Around the 10 digit range, things start to get noticably slower.

```
[is_prime_seive 1000000001;
 is_prime_seive 1000000003;
 is_prime_seive 1000000005;
 is_prime_seive 1000000007;
 is_prime_seive 1000000009;
 is_prime_seive 30000000001;]
```

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

1. Miller Rabin

   It could maybe be faster to implement a Miller-Rabin primality test, using a witness list long enough to guarantee determinism for 64-bit integers.

   This is a working (ish) Miller-Rabin implementation. However, it fails for large-ish inputs because (I think) of the power and powermod functions. It says ‘is_prime 1_000_000_009‘ is false, but this should be true. One of the intermediate computations in that call is ‘powermod 11 125_000_001 1_000_000_009‘, which returns a giant negative number, and it should not. I think that sometimes the expressions ‘r*r‘ or ‘a*r*r‘ inside of powermod have integer overflow. Maybe re-write this using ‘Zarith‘ or some other multiprecision library?

   This can be made to work with Zarith in utop. But for some reason, tuareg complains when using Zarith. Probably not worth fixing here.

```
let is_prime n =
  let small_primes = [2;3;5;7;11;13;17;19;23;29;31;37] in
  let admits_small_divisor n =
    let rec trial_division plist n = match plist with
      | [] -> false
      | p :: ps -> (n mod p = 0 || trial_division ps n) in
    trial_division small_primes n in
  match n with
  | n when n < 2 -> false
  | 2 -> true
  | n when n mod 2 = 0 -> false
  | n when List.mem n small_primes -> true
  | n -> if admits_small_divisor n
       then false
       else let rec range a b = match b-a with
              | 0 -> [a]
              | _ -> a :: range (a+1) b in
            let (--) a b = range a b in
            let rec power a b = match b with
              | 0 -> 1
              | 1 -> a
              | b -> let r = power a (b/2) in
                     if b mod 2 = 0
                     then r*r
                     else a*r*r in
            let rec powermod a b n = match b with
              | 0 -> 1
              | 1 -> a mod n
              | b -> let r = powermod a (b/2) n in
                     if b mod 2 = 0
                     then (r*r) mod n
                     else (a*r*r) mod n in
            let rec twos_power m =
              if m mod 2 = 1
              then 0
              else 1 + twos_power (m/2) in
            let s = twos_power (n-1) in
            let q = (n-1)/(power 2 s) in
            let psuedoprime_to_base_a a =
              let powerlist = List.map (function i -> powermod a (q*power 2 i)
              ↪ n) (0 -- (s-1)) in
              (List.hd powerlist = 1 || List.mem (n-1) powerlist) in
            not (List.mem false (List.map psuedoprime_to_base_a small_primes))
```

2. Elliptic Curve Primality ??

   Is this achievable using vanilla ocaml or reasonable libraries? Might be interesting to try.

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.

Euclidean algorithm.

```
let rec gcd a1 b1 =
  let a = if a1 < 0 then -a1 else a1 in
  let b = if b1 < 0 then -b1 else b1 in
  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;;
```

```
val gcd : int -> int -> int = <fun>
```

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

```
- : int = 648
```

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 totient function $\varphi(m)$ is defined as the number of positive integers $1 \leqslant r \leqslant m$ that are coprime to m.

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);;
```

```
val phi : int -> int = <fun>
```

```
phi 12321
```

```
- : int = 7992
```

To "find out" what $\varphi(p)$ is when p is prime, do the obvious numerical experiment.

```
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;;

let (--) a b = range a b;;

1 -- 100
|> List.filter is_prime_seive
|> List.map (fun p -> (p, phi p ))
```

```
- : (int * int) list =
[(2, 1); (3, 2); (5, 4); (7, 6); (11, 10); (13, 12); (17, 16); (19, 18);
 (23, 22); (29, 28); (31, 30); (37, 36); (41, 40); (43, 42); (47, 46);
 (53, 52); (59, 58); (61, 60); (67, 66); (71, 70); (73, 72); (79, 78);
 (83, 82); (89, 88); (97, 96)]
```

Numerical evidence that \$$\varphi$(p) = p-1.\$

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

Construct a flat list containing the prime factors in ascending order. Again, this is a naive approach. Using the `range` operator `--` from a previous problem to avoid too much repeated code.

```
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 candidates = (2 -- bound)
                        |> List.filter is_prime_seive in
       let rec smallest_prime_divisor lst m = match lst with
         | [] -> m
         | p :: ps -> if (m mod p = 0)
                      then (p)
                      else (smallest_prime_divisor ps m) in
       let p = smallest_prime_divisor candidates n in
       let q = n / p in
       p :: factors q;;
```

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

Various tests:

```
[factors 4;
 factors 5;
 factors 100;
 factors (17389*17389);
 factors (2*3*4*5*6*7*8*9*10*11*12*13)]
```

```
- : int list list =
[[2; 2]; [5]; [2; 2; 5; 5]; [17389; 17389];
 [2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 3; 3; 3; 3; 3; 5; 5; 7; 11; 13]]
```

It seems to work.

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: just take the prime factors from the previous problem and compress the list.

```
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 DONE 35 Calculate Euler's totient function (improved)

```
let eulerphi n =
  if n = 1 then 1 else
    let facts = factors_with_multiplicity n in
    let rec exp a b = match b with
      | 0 -> 1
      | b -> if (b mod 2 = 0)
             then (let rt = exp a (b/2) in rt * rt)
             else (let rt = exp a (b/2) in a * rt * rt) 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;;
```

```
val eulerphi : int -> int = <fun>
```

Check that it agrees with the previous implementation:

```
(1--100) |> List.map (fun p -> eulerphi p - phi p)
```

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

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

```
let time_phi n =
  let t1 = Sys.time() in
  let p = phi n in
  let t2 = Sys.time() in
  let msg = "calculated phi "
            ^ string_of_int n
            ^ " = "
            ^ string_of_int p
            ^ " in "
            ^ (string_of_float (t2 -. t1))
            ^ " seconds" in
print_endline msg;;

let time_eulerphi n =
  let t1 = Sys.time() in
  let p = eulerphi n in
  let t2 = Sys.time() in
  let msg = "calculated eulerphi "
            ^ string_of_int n
            ^ " = "
            ^ string_of_int p
            ^ " in "
            ^ (string_of_float (t2 -. t1))
            ^ " seconds" in
  print_endline msg;;
```

```
val time_eulerphi : int -> unit = <fun>
```

Now, timing the naive `phi` implementation on a large input

```
time_phi 142814;;
```

```
calculated phi 142814 = 60600 in 0.040892 seconds
- : unit = ()
```

But using the implementation that exploits multiplicativity of the φ function:

```
time_eulerphi 142814;;
```

```
calculated eulerphi 142814 = 60600 in 0.000310000000001 seconds
- : unit = ()
```

It's significantly faster.

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 =
 a -- b
 |> List.filter is_prime_seive;;
```

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

Check with Mathematica. Not a proof of correctness, but strong evidence.

```
List.length (all_primes 2 7920)
```

```
- : int = 1000
```

1.2.10 DONE 38 Goldbach's conjecture

Goldbach's conjecture says that every positive even number greater than 2 is the sum of two prime numbers. Example: $28 = 5 + 23$. It is one of the most famous conjectures in number theory that has not yet been proven. It has been numerically confirmed up to very large numbers. Write a function to find the two prime numbers that sum up to a given even integer.

```
let goldbach n =
if (n < 4 || n mod 2 = 1) then (0,0)
else let candidates = all_primes 1 n in
   let rec first_pair lst m = match lst with
     | [] -> (0,0)
     | p :: ps -> if (is_prime_seive (m-p))
                  then (p,m-p)
                  else (first_pair ps m) in
   first_pair candidates n
```

```
val goldbach : int -> int * int = <fun>
```

Run it on all even numbers up to 100:

```
(2--50)
|> List.map (fun m -> (2*m),goldbach (2*m))
```

```
- : (int * (int * int)) list =
[(4, (2, 2)); (6, (3, 3)); (8, (3, 5)); (10, (3, 7)); (12, (5, 7));
 (14, (3, 11)); (16, (3, 13)); (18, (5, 13)); (20, (3, 17)); (22, (3, 19));
 (24, (5, 19)); (26, (3, 23)); (28, (5, 23)); (30, (7, 23)); (32, (3, 29));
 (34, (3, 31)); (36, (5, 31)); (38, (7, 31)); (40, (3, 37)); (42, (5, 37));
 (44, (3, 41)); (46, (3, 43)); (48, (5, 43)); (50, (3, 47)); (52, (5, 47));
 (54, (7, 47)); (56, (3, 53)); (58, (5, 53)); (60, (7, 53)); (62, (3, 59));
 (64, (3, 61)); (66, (5, 61)); (68, (7, 61)); (70, (3, 67)); (72, (5, 67));
 (74, (3, 71)); (76, (3, 73)); (78, (5, 73)); (80, (7, 73)); (82, (3, 79));
 (84, (5, 79)); (86, (3, 83)); (88, (5, 83)); (90, (7, 83)); (92, (3, 89));
 (94, (5, 89)); (96, (7, 89)); (98, (19, 79)); (100, (3, 97))]
```

1.2.11 DONE 39 A list of Goldbach compositions

```
let goldbach_list n =
  if (n < 4 || n mod 2 = 1) then []
  else let candidates = all_primes 1 (n/2) in
       let rec all_pairs lst m = match lst with
         | [] -> []
         | p :: ps -> if (is_prime_seive (m-p))
                      then (p,m-p) :: all_pairs ps m
                      else (all_pairs ps m) in
       all_pairs candidates n
```

```
val goldbach_list : int -> (int * int) list = <fun>
```

Quick check.

```
goldbach_list 1000
```

```
- : (int * int) list =
[(3, 997); (17, 983); (23, 977); (29, 971); (47, 953); (53, 947); (59, 941);
 (71, 929); (89, 911); (113, 887); (137, 863); (173, 827); (179, 821);
 (191, 809); (227, 773); (239, 761); (257, 743); (281, 719); (317, 683);
 (347, 653); (353, 647); (359, 641); (383, 617); (401, 599); (431, 569);
 (443, 557); (479, 521); (491, 509)]
```

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.

(This problem is worded so vaguely…)

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

```
val gray : int -> string list = <fun>
```

Small test:

```
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));;
```

```
val cbal_tree : int -> char binary_tree list = <fun>
```

Small examples:

```
[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`, so probably correct

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;;
```

```
val is_symmetric : 'a binary_tree -> bool = <fun>
```

```
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;;
```

```
val construct : 'a list -> 'a binary_tree = <fun>
```

```
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.

Generate them all, then filter out the symmetric ones:

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

```
val sym_cbal_tree : int -> char binary_tree list = <fun>
```

Here they are when $n = 5$:

```
sym_cbal_tree 5;;
```

```
- : char binary_tree list =
[Node ('x', Node ('x', Empty, Node ('x', Empty, Empty)),
  Node ('x', Node ('x', Empty, Empty), Empty));
 Node ('x', Node ('x', Node ('x', Empty, Empty), Empty),
  Node ('x', Empty, Node ('x', Empty, Empty)))]
```

How many are there when $n = 57$?

```
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. We can verify that:

```
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]
```

How many are there for odd values of n? Here's a list of tuples, first entry is n, second is the number of symmetric completely balanced trees with n 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 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 determines `right`. Test this conjecture numerically to see that they definitely look the same.

```
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]
```

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]
```

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]
```

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]
```

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);;
```

```
val leaves : 'a binary_tree -> 'a list = <fun>
```

A small test:

```
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);;
```

```
val count_leaves : 'a binary_tree -> int = <fun>
```

A few small tests:

```
[Empty; Node('x',Node('y',Empty,Empty),Empty);t]
|> List.map count_leaves
```

```
- : int list = [0; 1; 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)));;
```

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

Small test:

```
range 0 6
|> List.map (at_level t)
```

```
- : char list list =
[[]; ['0']; ['1'; '2']; ['6'; '3'; '4']; ['7'; '8'; '5']; ['9']; []]
```

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);;
```

```
val internals : 'a binary_tree -> 'a list = <fun>
```

```
internals t
```

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

1.4.11 TODO 54

1.4.12 TODO 55

1.4.13 TODO 56

1.4.14 TODO 57

1.4.15 TODO 58

1.4.16 TODO 59

1.4.17 TODO 60

1.5 Multiway trees [5/5]

1.5.1 DONE 61 Count the nodes of a multiway tree

You need the type definition from the next problem to do this one. Here it is.

```
type 'a mult_tree = T of 'a * 'a mult_tree list;;
```

```
type 'a mult_tree = T of 'a * 'a mult_tree list
```

Count nodes in the obvious recursive way.

```
let rec count_nodes =
  function T (r, lst) ->
    let sum lst =
      let rec sum_aux p lst = match lst with
        | [] -> p
        | x :: xs -> sum_aux (p+x) xs in
      sum_aux 0 lst in
    lst
    |> List.map count_nodes
    |> sum
    |> (+) 1
```

```
val count_nodes : 'a mult_tree -> int = <fun>
```

Using the example from the problem page:

```
let ex = T ('a', [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e',
↪  [])])]) in
    count_nodes ex;;
```

```
- : int = 7
```

1.5.2 DONE 62 Tree construction from a node string

A multiway tree is composed of a root element and a (possibly empty) set of successors which are multiway trees themselves. A multiway tree is never empty. The set of successor trees is sometimes called a forest.

To represent multiway trees, we will use the following type which is a direct translation of the definition:

```
type 'a mult_tree = T of 'a * 'a mult_tree list;;
```

The example tree depicted opposite is therefore represented by the following OCaml expression:

```
T ('a', [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e', [])])]);;
```

We suppose that the nodes of a multiway tree contain single characters. In the depth-first order sequence of its nodes, a special character ^ has been inserted whenever, during the tree traversal, the move is a backtrack to the previous level.

By this rule, the tree in the figure opposite is represented as: `afg^^c^bd^e^^^`.

Write functions `string_of_tree : char mult_tree -> string` to construct the string representing the tree and `tree_of_string : string -> char mult_tree` to construct the tree when the string is given.

I'll do `string_of_tree` first because I suspect it's the easier of the two:

```
let rec string_of_tree t = match t with
  | T (x, []) -> Char.escaped x
  | T (x, lst) -> let append_caret =
                    (fun s -> s ^ "^") in
                  lst
                  |> List.map string_of_tree
                  |> List.map append_caret
                  |> List.fold_left (^) ""
                  |> (^) (Char.escaped x)
```

```
val string_of_tree : char mult_tree -> string = <fun>
```

Test on their example:

```
let t = T ('a', [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e',
↪  [])])]) in
    string_of_tree t
```

```
- : string = "afg^^c^bd^e^^"
```

Now for `tree_of_string`. I don't think using imperative stuff in the string processing was totally necessary but it seems to work fine.

```ocaml
let rec tree_of_string s =
  let n = String.length s in
  let first = String.get s 0 in
  let rest = String.sub s 1 (n - 1) in
  let rec split_into_substrings s = match s with
    | "" -> []
    | s -> let tally = ref 1 in
           let idx = ref 0 in
           while !tally > 0
           do (idx := !idx + 1;
               if (Char.escaped s.[!idx] <> "^")
               then (tally := !tally + 1)
               else (tally := !tally - 1);)
           done;
           let n = String.length s in
           let first_substring = String.sub s 0 (!idx) in
           let rest = String.sub s (!idx + 1) (n - !idx - 1) in
           first_substring :: (split_into_substrings rest) in
  match split_into_substrings rest with
  | [] -> T (first, [])
  | _ -> T (first, rest
                |> split_into_substrings
                |> List.map tree_of_string);;
```

```
val tree_of_string : string -> char mult_tree = <fun>
```

Test with their example

```ocaml
tree_of_string "afg^^c^bd^e^^"
```

```
- : char mult_tree =
T ('a',
 [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e', [])])])
```

A few tests to check that one is the inverse of the other:

```ocaml
["a";
 "abc^^";
 "abf^g^^chk^^^dil^mn^^^^ej^^"]
|> List.map tree_of_string
|> List.map string_of_tree
```

```
- : string list = ["a"; "abc^^"; "abf^g^^chk^^^dil^mn^^^^ej^^"]
```

1.5.3 DONE 63 Determine the internal path length of a tree

We define the internal path length of a multiway tree as the total sum of the path lengths from the root to all nodes of the tree. By this definition, the tree `t` in the figure of the previous problem has an internal path length of 9. Write a function `ipl tree` that returns the internal path length of `tree`.

```
let rec ipl tree =
  let rec ipl_aux d tree = match tree with
    | T (x, []) -> d
    | T (x, lst) -> let sum lst =
                      let rec sum_aux p lst = match lst with
                        | [] -> p
                        | n :: ns -> sum_aux (p + n) ns in
                      sum_aux 0 lst in
                    d + (lst |> List.map (ipl_aux (d + 1)) |> sum) in
ipl_aux 0 tree;;
```

```
val ipl : 'a mult_tree -> int = <fun>
```

test on the given example tree:

```
let ex = T ('a', [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e',
↪  [])])]) in
    ipl ex;;
```

```
- : int = 9
```

1.5.4 DONE 64 Construct the bottom-up order sequence of the tree nodes

Write a function `bottom_up t` which constructs the bottom-up sequence of the nodes of the multiway tree `t`.

```
let rec bottom_up t = match t with
  | T (x, []) -> [x]
  | T (x, lst) -> (lst
                  |> List.map bottom_up
                  |> List.fold_left (@) []) @ [x];;
```

```
val bottom_up : 'a mult_tree -> 'a list = <fun>
```

Test on their example list:

```
let ex = T ('a', [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e',
↪  [])])]) in
    bottom_up ex;;
```

```
- : char list = ['g'; 'f'; 'c'; 'd'; 'e'; 'b'; 'a']
```

1.5.5 DONE 65 Lisp-like tree representation

```
let rec lispy =
  function T (c, lst) ->
            let s = Char.escaped c in
            match lst with
            | [] -> s
            | _ -> lst
                   |> List.map lispy
                   |> List.fold_left (fun a b -> a^" "^b) ""
                   |> fun i -> "("^s^i^")";;
```

```
val lispy : char mult_tree -> string = <fun>
```

Test the given examples:

```
[lispy (T ('a', []));
 lispy (T ('a', [T ('b', [])]));
 lispy (T ('a', [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e',
 ↪  [])])]))]
```

```
- : string list = ["a"; "(a b)"; "(a (f g) c (b d e))"]
```

1.6 Graphs [0/11]

1.6.1 TODO 66

1.6.2 TODO 67

1.6.3 TODO 68

1.6.4 TODO 69

1.6.5 TODO 70

1.6.6 TODO 71

1.6.7 TODO 72

1.6.8 TODO 73

1.6.9 TODO 74

1.6.10 TODO 75

1.6.11 TODO 76

1.7 Miscellaneous [0/9]

1.7.1 TODO 77

1.7.2 TODO 78

1.7.3 TODO 79

1.7.4 TODO 80

1.7.5 TODO 81

1.7.6 TODO 82

1.7.7 TODO 83

1.7.8 TODO 84

1.7.9 TODO 85