Your Help is Needed
Many of the solutions below have been written by Victor Nicollet. Please contribute more solutions or improve the existing ones.

99 Problems (solved) in OCaml

This section is inspired by Ninety-Nine Lisp Problems which in turn was based on "Prolog problem list". For each of these questions, some simple tests are

shown—they may also serve to make the question clearer if needed. To work on these problems, we recommend you first install OCaml or use it inside your browser. The source of the following problems is available on GitHub.

Working with lists

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

Solution

```
# let rec last = function
    | [] -> None
    | [x] -> Some x
    | _ :: t -> last t;;
val last : 'a list -> 'a option = <fun>

# last [ "a" ; "b" ; "c" ; "d" ];;
- : string option = Some "d"
# last [];;
- : 'a option = None
```

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

Solution

```
# let rec last_two = function
    | [] | [_] -> None
    | [x;y] -> Some (x,y)
    | _::t -> last_two t;;
val last_two : 'a list -> ('a * 'a) option = <fun>

# last_two [ "a" ; "b" ; "c" ; "d" ];;
- : (string * string) option = Some ("c", "d")
# last_two [ "a" ];;
- : (string * string) option = None
```

Find the `k`'th element of a list. (*easy*)

Solution

```
# let rec at k = function
    | [] -> None
    | h :: t -> if k = 1 then Some h else at (k-1) t;;
val at : int -> 'a list -> 'a option = <fun>

# at 3 [ "a" ; "b"; "c"; "d"; "e" ];;
- : string option = Some "c"
```

```
# at 3 [ "a" ];;
- : string option = None
```

Find the number of elements of a list. (*easy*)

OCaml standard library has `List.length` but we ask that you reimplement it. Bonus for a tail recursive solution.

Solution

```
# (* This function is tail-recursive: it uses a constant amount of
     stack memory regardless of list size. *)
  let length list =
    let rec aux n = function
      | [] -> n
      | _::t -> aux (n+1) t
    in aux 0 list;;
val length : 'a list -> int = <fun>

# length [ "a" ; "b" ; "c"];;
- : int = 3
# length [];;
- : int = 0
```

Reverse a list. (*easy*)

OCaml standard library has `List.rev` but we ask that you reimplement it.

Solution

```
# let rev list =
    let rec aux acc = function
      | [] -> acc
      | h::t -> aux (h::acc) t in
    aux [] list;;
val rev : 'a list -> 'a list = <fun>

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

Find out whether a list is a palindrome. (*easy*)

HINT: a palindrome is its own reverse.

Solution

```
# let is_palindrome list =
    list = List.rev list
```

```
  (* One can use either the rev function from the previous problem, or the
     built-in List.rev *);;
val is_palindrome : 'a list -> bool = <fun>

# is_palindrome [ "x" ; "a" ; "m" ; "a" ; "x" ];;
- : bool = true
# not (is_palindrome [ "a" ; "b" ]);;
- : bool = true
```

Flatten a nested list structure. (*medium*)

```
# (* There is no nested list type in OCaml, so we need to define one
     first. A node of a nested list is either an element, or a list of
     nodes. *)
  type 'a node =
    | One of 'a
    | Many of 'a node list;;
type 'a node = One of 'a | Many of 'a node list
```

Solution

```
# (* This function traverses the list, prepending any encountered elements
    to an accumulator, which flattens the list in inverse order. It can
    then be reversed to obtain the actual flattened list. *)

  let flatten list =
    let rec aux acc = function
      | [] -> acc
      | One x :: t -> aux (x :: acc) t
      | Many l :: t -> aux (aux acc l) t in
    List.rev (aux [] list);;
val flatten : 'a node list -> 'a list = <fun>

# flatten [ One "a" ; Many [ One "b" ; Many [ One "c" ; One "d" ] ; One "e" ] ];;
- : string list = ["a"; "b"; "c"; "d"; "e"]
```

Eliminate consecutive duplicates of list elements. (*medium*)

Solution

```
# let rec compress = function
    | a :: (b :: _ as t) -> if a = b then compress t else a :: compress t
    | smaller -> smaller;;
val compress : 'a list -> 'a list = <fun>

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

Pack consecutive duplicates of list elements into sublists. (*medium*)

Solution

```
# let pack list =
    let rec aux current acc = function
      | [] -> []    (* Can only be reached if original list is empty *)
      | [x] -> (x :: current) :: acc
      | a :: (b :: _ as t) ->
         if a = b then aux (a :: current) acc t
         else aux [] ((a :: current) :: acc) t  in
    List.rev (aux [] [] list);;
val pack : 'a list -> 'a list list = <fun>

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

Run-length encoding of a list. (*easy*)

If you need so, refresh your memory about run-length encoding.

Solution

```
# let encode list =
    let rec aux count acc = function
      | [] -> [] (* Can only be reached if original list is empty *)
      | [x] -> (count+1, x) :: acc
      | a :: (b :: _ as t) -> if a = b then aux (count + 1) acc t
                              else aux 0 ((count+1,a) :: acc) t in
    List.rev (aux 0 [] list);;
val encode : 'a list -> (int * 'a) list = <fun>
```

An alternative solution, which is shorter but requires more memory, is to use the [pack](#Packconsecutiveduplicatesoflistelementsintosublistsmedium) function declared above:

```
# let encode list =
    List.map (fun l -> (List.length l, List.hd l)) (pack list);;
val encode : 'a list -> (int * 'a) list = <fun>
```

Here is an example:

```
# encode ["a";"a";"a";"a";"b";"c";"c";"a";"a";"d";"e";"e";"e";"e"];;
- : (int * string) list =
[(4, "a"); (1, "b"); (2, "c"); (2, "a"); (1, "d"); (4, "e")]
```

Modified run-length encoding. (*easy*)

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
```

Solution

```
# let encode l =
    let create_tuple cnt elem =
      if cnt = 1 then One elem
      else Many (cnt, elem) in
    let rec aux count acc = function
      | [] -> []
      | [x] -> (create_tuple (count+1) x) :: acc
      | hd :: (snd :: _ as tl) ->
          if hd = snd then aux (count + 1) acc tl
          else aux 0 ((create_tuple (count + 1) hd) :: acc) tl in
      List.rev (aux 0 [] l);;
val encode : 'a list -> 'a rle list = <fun>
```

```
# encode ["a";"a";"a";"a";"b";"c";"c";"a";"a";"d";"e";"e";"e";"e"];;
- : string rle list =
[Many (4, "a"); One "b"; Many (2, "c"); Many (2, "a"); One "d";
 Many (4, "e")]
```

Decode a run-length encoded list. (*medium*)

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

Solution

```
# let decode list =
    let rec many acc n x =
      if n = 0 then acc else many (x :: acc) (n-1) x in
    let rec aux acc = function
      | [] -> acc
      | One x :: t -> aux (x :: acc) t
      | Many (n,x) :: t -> aux (many acc n x) t  in
    aux [] (List.rev list);;
val decode : 'a rle list -> 'a list = <fun>
```

```
# decode [Many (4,"a"); One "b"; Many (2,"c"); Many (2,"a"); One "d"; Many (4,"e")];;
- : string list =
["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "e"; "e"; "e"; "e"]
```

Run-length encoding of a list (direct solution). (*medium*)

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.

Solution

```
# let encode list =
    let rle count x = if count = 0 then One x else Many (count + 1, x) in
    let rec aux count acc = function
      | [] -> [] (* Can only be reached if original list is empty *)
      | [x] -> rle count x :: acc
      | a :: (b :: _ as t) -> if a = b then aux (count + 1) acc t
                              else aux 0 (rle count a :: acc) t  in
    List.rev (aux 0 [] list);;
val encode : 'a list -> 'a rle list = <fun>

# encode ["a";"a";"a";"a";"b";"c";"c";"a";"a";"d";"e";"e";"e";"e"];;
- : string rle list =
[Many (4, "a"); One "b"; Many (2, "c"); Many (2, "a"); One "d";
 Many (4, "e")]
```

Duplicate the elements of a list. (*easy*)

Solution

```
# let rec duplicate = function
    | [] -> []
    | h :: t -> h :: h :: duplicate t;;
val duplicate : 'a list -> 'a list = <fun>
```

Remark: this function is not tail recursive. Can you modify it so it becomes so?

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

Replicate the elements of a list a given number of times. (*medium*)

Solution

```
# let replicate list n =
    let rec prepend n acc x =
      if n = 0 then acc else prepend (n-1) (x :: acc) x in
    let rec aux acc = function
      | [] -> acc
      | h :: t -> aux (prepend n acc h) t  in
```

```
    (* This could also be written as:
       List.fold_left (prepend n) [] (List.rev list) *)
    aux [] (List.rev list);;
val replicate : 'a list -> int -> 'a list = <fun>
```

Note that `List.rev list` is needed only because we want `aux` to be [tail recursive](http://en.wikipedia.org/wiki/Tail_call).

```
# replicate ["a";"b";"c"] 3;;
- : string list = ["a"; "a"; "a"; "b"; "b"; "b"; "c"; "c"; "c"]
```

Drop every N'th element from a list. (*medium*)

Solution

```
# let drop list n =
    let rec aux i = function
      | [] -> []
      | h :: t -> if i = n then aux 1 t else h :: aux (i+1) t  in
    aux 1 list;;
val drop : 'a list -> int -> 'a list = <fun>

# drop ["a";"b";"c";"d";"e";"f";"g";"h";"i";"j"] 3;;
- : string list = ["a"; "b"; "d"; "e"; "g"; "h"; "j"]
```

Split a list into two parts; the length of the first part is given. (*easy*)

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.

Solution

```
# let split list n =
    let rec aux i acc = function
      | [] -> List.rev acc, []
      | h :: t as l -> if i = 0 then List.rev acc, l
                       else aux (i-1) (h :: acc) t  in
    aux n [] list;;
val split : 'a list -> int -> 'a list * 'a list = <fun>

# split ["a";"b";"c";"d";"e";"f";"g";"h";"i";"j"] 3;;
- : string list * string list =
(["a"; "b"; "c"], ["d"; "e"; "f"; "g"; "h"; "i"; "j"])
# split ["a";"b";"c";"d"] 5;;
- : string list * string list = (["a"; "b"; "c"; "d"], [])
```

Extract a slice from a list. (*medium*)

Given two indices, `i` and `k`, the slice is the list containing the elements between the `i`'th and `k`'th element of the original list (both limits included). Start counting

the elements with 0 (this is the way the `List` module numbers elements).

Solution

```
# let slice list i k =
    let rec take n = function
      | [] -> []
      | h :: t -> if n = 0 then [] else h :: take (n-1) t
    in
    let rec drop n = function
      | [] -> []
      | h :: t as l -> if n = 0 then l else drop (n-1) t
    in
    take (k - i + 1) (drop i list);;
val slice : 'a list -> int -> int -> 'a list = <fun>
```

This solution has a drawback, namely that the `take` function is not [tail recurvive](https://en.wikipedia.org/wiki/Tail_call) so it may exhaust the stack when given a very long list. You may also notice that the structure of `take` and `drop` is similar and you may want to abstract their common skeleton in a single function. Here is a solution.

```
# let rec fold_until f acc n = function
    | [] -> (acc, [])
    | h :: t as l -> if n = 0 then (acc, l)
                     else fold_until f (f acc h) (n-1) t

  let slice list i k =
    let _, list = fold_until (fun _ _ -> []) [] i list in
    let taken, _ = fold_until (fun acc h -> h :: acc) [] (k - i + 1) list in
    List.rev taken;;
val fold_until : ('a -> 'b -> 'a) -> 'a -> int -> 'b list -> 'a * 'b list =
  <fun>
val slice : 'a list -> int -> int -> 'a list = <fun>

# slice ["a";"b";"c";"d";"e";"f";"g";"h";"i";"j"] 2 6;;
- : string list = ["c"; "d"; "e"; "f"; "g"]
```

Rotate a list N places to the left. (*medium*)

Solution

```
# let split list n =
    let rec aux i acc = function
      | [] -> List.rev acc, []
      | h :: t as l -> if i = 0 then List.rev acc, l
                       else aux (i-1) (h :: acc) t  in
    aux n [] list

  let rotate list n =
    let len = List.length list in
```

```
    (* Compute a rotation value between 0 and len-1 *)
    let n = if len = 0 then 0 else (n mod len + len) mod len in
    if n = 0 then list
    else let a, b = split list n in b @ a;;
val split : 'a list -> int -> 'a list * 'a list = <fun>
val rotate : 'a list -> int -> 'a list = <fun>

# rotate ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"] 3;;
- : string list = ["d"; "e"; "f"; "g"; "h"; "a"; "b"; "c"]
# rotate ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"] (-2);;
- : string list = ["g"; "h"; "a"; "b"; "c"; "d"; "e"; "f"]
```

Remove the K'th element from a list. (*easy*)

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

Solution

```
# let rec remove_at n = function
    | [] -> []
    | h :: t -> if n = 0 then t else h :: remove_at (n-1) t;;
val remove_at : int -> 'a list -> 'a list = <fun>

# remove_at 1 ["a";"b";"c";"d"];;
- : string list = ["a"; "c"; "d"]
```

Insert an element at a given position into a list. (*easy*)

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.)

Solution

```
# let rec insert_at x n = function
    | [] -> [x]
    | h :: t as l -> if n = 0 then x :: l else h :: insert_at x (n-1) t;;
val insert_at : 'a -> int -> 'a list -> 'a list = <fun>

# insert_at "alfa" 1 ["a";"b";"c";"d"];;
- : string list = ["a"; "alfa"; "b"; "c"; "d"]
# insert_at "alfa" 3 ["a";"b";"c";"d"];;
- : string list = ["a"; "b"; "c"; "alfa"; "d"]
# insert_at "alfa" 4 ["a";"b";"c";"d"];;
- : string list = ["a"; "b"; "c"; "d"; "alfa"]
```

Create a list containing all integers within a given range. (*easy*)

If first argument is smaller than second, produce a list in decreasing order.

Solution

```
# let range a b =
    let rec aux a b =
      if a > b then [] else a :: aux (a+1) b  in
    if a > b then List.rev (aux b a) else aux a b;;
val range : int -> int -> int list = <fun>

# range 4 9;;
- : int list = [4; 5; 6; 7; 8; 9]
# range 9 4;;
- : int list = [9; 8; 7; 6; 5; 4]
```

Extract a given number of randomly selected elements from a list. (*medium*)

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.

Solution

```
# let rec rand_select list n =
    let rec extract acc n = function
      | [] -> raise Not_found
      | h :: t -> if n = 0 then (h, acc @ t) else extract (h::acc) (n-1) t
    in
    let extract_rand list len =
      extract [] (Random.int len) list
    in
    let rec aux n acc list len =
      if n = 0 then acc else
        let picked, rest = extract_rand list len in
        aux (n-1) (picked :: acc) rest (len-1)
    in
    let len = List.length list in
    aux (min n len) [] list len;;
val rand_select : 'a list -> int -> 'a list = <fun>

# rand_select ["a";"b";"c";"d";"e";"f";"g";"h"] 3;;
- : string list = ["g"; "d"; "a"]
```

Lotto: Draw N different random numbers from the set 1..M. (*easy*)

The selected numbers shall be returned in a list.

Solution

```
# (* [range] and [rand_select] defined in problems above *)
  let lotto_select n m = rand_select (range 1 m) n;;
```

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

# lotto_select 6 49;;
- : int list = [10; 20; 44; 22; 41; 2]
```

Generate a random permutation of the elements of a list. (*easy*)

Solution

```
# let rec permutation list =
    let rec extract acc n = function
      | [] -> raise Not_found
      | h :: t -> if n = 0 then (h, acc @ t) else extract (h::acc) (n-1) t
    in
    let extract_rand list len =
      extract [] (Random.int len) list
    in
    let rec aux acc list len =
      if len = 0 then acc else
        let picked, rest = extract_rand list len in
        aux (picked :: acc) rest (len-1)
    in
    aux [] list (List.length list);;
val permutation : 'a list -> 'a list = <fun>

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

Generate the combinations of K distinct objects chosen from the N elements of a list. (*medium*)

In how many ways can a committee of 3 be chosen from a group of 12 people? We all know that there are C(12,3) = 220 possibilities (C(N,K) denotes the well-known binomial coefficients). For pure mathematicians, this result may be great. But we want to really generate all the possibilities in a list.

Solution

```
# let extract k list =
    let rec aux k acc emit = function
      | [] -> acc
      | h :: t ->
        if k = 1 then aux k (emit [h] acc) emit t else
          let new_emit x = emit (h :: x) in
          aux k (aux (k-1) acc new_emit t) emit t
    in
    let emit x acc = x :: acc in
    aux k [] emit list;;
val extract : int -> 'a list -> 'a list list = <fun>
```

```
# extract 2 ["a";"b";"c";"d"];;
- : string list list =
[["c"; "d"]; ["b"; "d"]; ["b"; "c"]; ["a"; "d"]; ["a"; "c"]; ["a"; "b"]]
```

Group the elements of a set into disjoint subsets. (*medium*)

1. 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.
2. 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.

Solution

```
# (* This implementation is less streamlined than the one-extraction
    version, because more work is done on the lists after each
    transform to prepend the actual items. The end result is cleaner
    in terms of code, though. *)

    let group list sizes =
      let initial = List.map (fun size -> size, []) sizes in

    (* The core of the function. Prepend accepts a list of groups,
       each with the number of items that should be added, and
       prepends the item to every group that can support it, thus
       turning [1,a ; 2,b ; 0,c] into [ [0,x::a ; 2,b ; 0,c ];
       [1,a ; 1,x::b ; 0,c]; [ 1,a ; 2,b ; 0,c ]]

       Again, in the prolog language (for which these questions are
       originally intended), this function is a whole lot simpler.  *)
    let prepend p list =
      let emit l acc = l :: acc in
      let rec aux emit acc = function
        | [] -> emit [] acc
        | (n,l) as h :: t ->
           let acc = if n > 0 then emit ((n-1, p::l) :: t) acc
                     else acc in
           aux (fun l acc -> emit (h :: l) acc) acc t
      in
      aux emit [] list
    in
    let rec aux = function
      | [] -> [ initial ]
      | h :: t -> List.concat (List.map (prepend h) (aux t))
    in
    let all = aux list in
    (* Don't forget to eliminate all group sets that have non-full
       groups *)
    let complete = List.filter (List.for_all (fun (x,_) -> x = 0)) all in
    List.map (List.map snd) complete;;
val group : 'a list -> int list -> 'a list list list = <fun>
```

```
# group ["a";"b";"c";"d"] [2;1];;
- : string list list list =
[[["a"; "b"]; ["c"]]; [["a"; "c"]; ["b"]]; [["b"; "c"]; ["a"]];
 [["a"; "b"]; ["d"]]; [["a"; "c"]; ["d"]]; [["b"; "c"]; ["d"]];
 [["a"; "d"]; ["b"]]; [["b"; "d"]; ["a"]]; [["a"; "d"]; ["c"]];
 [["b"; "d"]; ["c"]]; [["c"; "d"]; ["a"]]; [["c"; "d"]; ["b"]]]
```

Sorting a list of lists according to length of sublists. (*medium*)

1. 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.

2. 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.

Solution

```
# (* We might not be allowed to use built-in List.sort, so here's an
     eight-line implementation of insertion sort – O(n²) time
     complexity. *)
  let rec insert cmp e = function
    | [] -> [e]
    | h :: t as l -> if cmp e h <= 0 then e :: l else h :: insert cmp e t

  let rec sort cmp = function
    | [] -> []
    | h :: t -> insert cmp h (sort cmp t)

  (* Sorting according to length : prepend length, sort, remove length *)
  let length_sort lists =
    let lists = List.map (fun list -> List.length list, list) lists in
    let lists = sort (fun a b -> compare (fst a) (fst b)) lists in
    List.map snd lists;;
val insert : ('a -> 'a -> int) -> 'a -> 'a list -> 'a list = <fun>
val sort : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
val length_sort : 'a list list -> 'a list list = <fun>
# (* Sorting according to length frequency : prepend frequency, sort,
     remove frequency. Frequencies are extracted by sorting lengths
     and applying RLE to count occurences of each length (see problem
     "Run-length encoding of a list.") *)
  let rle list =
    let rec aux count acc = function
      | [] -> [] (* Can only be reached if original list is empty *)
      | [x] -> (x, count + 1) :: acc
      | a :: (b :: _ as t) ->
         if a = b then aux (count + 1) acc t
         else aux 0 ((a, count + 1) :: acc) t in
```

```
    aux 0 [] list

  let frequency_sort lists =
    let lengths = List.map List.length lists in
    let freq = rle (sort compare lengths) in
    let by_freq =
      List.map (fun list -> List.assoc (List.length list) freq , list) lists in
    let sorted = sort (fun a b -> compare (fst a) (fst b)) by_freq in
    List.map snd sorted;;
val rle : 'a list -> ('a * int) list = <fun>
val frequency_sort : 'a list list -> 'a list list = <fun>

# length_sort [ ["a";"b";"c"]; ["d";"e"]; ["f";"g";"h"]; ["d";"e"];
                ["i";"j";"k";"l"]; ["m";"n"]; ["o"] ];;
- : string list list =
[["o"]; ["d"; "e"]; ["d"; "e"]; ["m"; "n"]; ["a"; "b"; "c"]; ["f"; "g"; "h"];
 ["i"; "j"; "k"; "l"]]
# frequency_sort [ ["a";"b";"c"]; ["d";"e"]; ["f";"g";"h"]; ["d";"e"];
                   ["i";"j";"k";"l"]; ["m";"n"]; ["o"] ];;
- : string list list =
[["i"; "j"; "k"; "l"]; ["o"]; ["a"; "b"; "c"]; ["f"; "g"; "h"]; ["d"; "e"];
 ["d"; "e"]; ["m"; "n"]]
```

Arithmetic

Determine whether a given integer number is prime. (*medium*)

Solution

Recall that `d` divides `n` iff `n mod d = 0`. This is a naive solution. See the [Sieve of Eratosthenes](http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes) for a more clever one.

```
# let is_prime n =
    let n = abs n in
    let rec is_not_divisor d =
      d * d > n || (n mod d <> 0 && is_not_divisor (d+1)) in
    n <> 1 && is_not_divisor 2;;
val is_prime : int -> bool = <fun>

# not(is_prime 1);;
- : bool = true
# is_prime 7;;
- : bool = true
# not (is_prime 12);;
- : bool = true
```

Determine the greatest common divisor of two positive integer numbers. (*medium*)

Use Euclid's algorithm.

Solution

```
# let rec gcd a b =
    if b = 0 then a else gcd b (a mod b);;
val gcd : int -> int -> int = <fun>

# gcd 13 27;;
- : int = 1
# gcd 20536 7826;;
- : int = 2
```

Determine whether two positive integer numbers are coprime. (*easy*)

Two numbers are coprime if their greatest common divisor equals 1.

Solution

```
# (* [gcd] is defined in the previous question *)
  let coprime a b = gcd a b = 1;;
val coprime : int -> int -> bool = <fun>

# coprime 13 27;;
- : bool = true
# not (coprime 20536 7826);;
- : bool = true
```

Calculate Euler's totient function φ(m). (*medium*)

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

Find out what the value of φ(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).

Solution

```
# (* [coprime] is defined in the previous question *)
  let phi n =
    let rec count_coprime acc d =
      if d < n then
        count_coprime (if coprime n d then acc + 1 else acc) (d + 1)
      else acc
    in
```

```
    if n = 1 then 1 else count_coprime 0 1;;
val phi : int -> int = <fun>

# phi 10;;
- : int = 4
# phi 13;;
- : int = 12
```

Determine the prime factors of a given positive integer. (*medium*)

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

Solution

```
# (* Recall that d divides n iff [n mod d = 0] *)
  let factors n =
    let rec aux d n =
      if n = 1 then [] else
        if n mod d = 0 then d :: aux d (n / d) else aux (d+1) n
    in
    aux 2 n;;
val factors : int -> int list = <fun>

# factors 315;;
- : int list = [3; 3; 5; 7]
```

Determine the prime factors of a given positive integer (2). (*medium*)

Construct a list containing the prime factors and their multiplicity. *Hint:* The problem is similar to problem Run-length encoding of a list (direct solution).

Solution

```
# let factors n =
    let rec aux d n =
      if n = 1 then [] else
        if n mod d = 0 then
          match aux d (n / d) with
          | (h,n) :: t when h = d -> (h,n+1) :: t
          | l -> (d,1) :: l
        else aux (d+1) n
    in
    aux 2 n;;
val factors : int -> (int * int) list = <fun>

# factors 315;;
- : (int * int) list = [(3, 2); (5, 1); (7, 1)]
```

Calculate Euler's totient function φ(m) (improved). (*medium*)

See problem "Calculate Euler's totient function φ(m)" for the definition of Euler's totient function. If the list of the prime factors of a number m is known in the form of the previous problem then the function phi(m) can be efficiently calculated as follows: Let `[(p1, m1); (p2, m2); (p3, m3); ...]` be the list of prime factors (and their multiplicities) of a given number m. Then φ(m) can be calculated with the following formula:

$$\varphi(m) = (p1 - 1) \times p1^{m1 - 1} \times (p2 - 1) \times p2^{m2 - 1} \times (p3 - 1) \times p3^{m3 - 1} \times \ldots$$

Solution

```
# (* Naive power function. *)
  let rec pow n p = if p < 1 then 1 else n * pow n (p-1);;
val pow : int -> int -> int = <fun>
# (* [factors] is defined in the previous question. *)
  let phi_improved n =
    let rec aux acc = function
      | [] -> acc
      | (p,m) :: t -> aux ((p - 1) * pow p (m - 1) * acc) t in
    aux 1 (factors n);;
val phi_improved : int -> int = <fun>

# phi_improved 10;;
- : int = 4
# phi_improved 13;;
- : int = 12
```

Compare the two methods of calculating Euler's totient function. (*easy*)

Use the solutions of problems "Calculate Euler's totient function φ(m)" and "Calculate Euler's totient function φ(m) (improved)" to compare the algorithms. Take the number of logical inferences as a measure for efficiency. Try to calculate φ(10090) as an example.

Solution

```
# (* Naive [timeit] function.  It requires the [Unix] module to be loaded. *)
  let timeit f a =
    let t0 = Unix.gettimeofday() in
    ignore(f a);
    let t1 = Unix.gettimeofday() in
    t1 -. t0;;
val timeit : ('a -> 'b) -> 'a -> float = <fun>

# timeit phi 10090;;
- : float = 0.00589895248413085938
# timeit phi_improved 10090;;
- : float = 7.2956085205078125e-05
```

A list of prime numbers. (*easy*)

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

Solution

```
# let is_prime n =
    let n = max n (-n) in
    let rec is_not_divisor d =
      d * d > n || (n mod d <> 0 && is_not_divisor (d+1)) in
    is_not_divisor 2

  let rec all_primes a b =
    if a > b then [] else
      let rest = all_primes (a + 1) b in
      if is_prime a then a :: rest else rest;;
val is_prime : int -> bool = <fun>
val all_primes : int -> int -> int list = <fun>

# List.length (all_primes 2 7920);;
- : int = 1000
```

Goldbach's conjecture. (*medium*)

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 facts in number theory that has not been proved to be correct in the general case. 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.

Solution

```
# (* [is_prime] is defined in the previous solution *)
  let goldbach n =
    let rec aux d =
      if is_prime d && is_prime (n - d) then (d, n-d)
      else aux (d+1) in
    aux 2;;
val goldbach : int -> int * int = <fun>

# goldbach 28;;
- : int * int = (5, 23)
```

A list of Goldbach compositions. (*medium*)

Given a range of integers by its lower and upper limit, print a list of all even numbers and their Goldbach composition.

In most cases, if an even number is written as the sum of two prime numbers, one of them is very small. Very rarely, the primes are both bigger than say 50. Try to find out how many such cases there are in the range 2..3000.

Solution

```
# (* [goldbach] is defined in the previous question. *)
  let rec goldbach_list a b =
    if a > b then [] else
      if a mod 2 = 1 then goldbach_list (a+1) b
      else (a, goldbach a) :: goldbach_list (a+2) b

  let goldbach_limit a b lim =
    List.filter (fun (_,(a,b)) -> a > lim && b > lim) (goldbach_list a b);;
val goldbach_list : int -> int -> (int * (int * int)) list = <fun>
val goldbach_limit : int -> int -> int -> (int * (int * int)) list = <fun>

# goldbach_list 9 20;;
- : (int * (int * int)) list =
[(10, (3, 7)); (12, (5, 7)); (14, (3, 11)); (16, (3, 13)); (18, (5, 13));
 (20, (3, 17))]
# goldbach_limit 1 2000 50;;
- : (int * (int * int)) list =
[(992, (73, 919)); (1382, (61, 1321)); (1856, (67, 1789));
 (1928, (61, 1867))]
```

Logic and Codes

Let us define a small "language" for boolean expressions containing variables:

```
# type bool_expr =
    | Var of string
    | Not of bool_expr
    | And of bool_expr * bool_expr
    | Or of bool_expr * bool_expr;;
type bool_expr =
    Var of string
  | Not of bool_expr
  | And of bool_expr * bool_expr
  | Or of bool_expr * bool_expr
```

A logical expression in two variables can then be written in prefix notation. For example, `(a ∨ b) ∧ (a ∧ b)` is written:

```
# And(Or(Var "a", Var "b"), And(Var "a", Var "b"));;
- : bool_expr = And (Or (Var "a", Var "b"), And (Var "a", Var "b"))
```

Truth tables for logical expressions (2 variables). (*medium*)

Define a function, `table2` which returns the truth table of a given logical

expression in two variables (specified as arguments). The return value must be a list of triples containing `(value_of_a, balue_of_b, value_of_expr)`.

Solution

```
# let rec eval2 a val_a b val_b = function
    | Var x -> if x = a then val_a
               else if x = b then val_b
               else failwith "The expression contains an invalid variable"
    | Not e -> not(eval2 a val_a b val_b e)
    | And(e1, e2) -> eval2 a val_a b val_b e1 && eval2 a val_a b val_b e2
    | Or(e1, e2) -> eval2 a val_a b val_b e1 || eval2 a val_a b val_b e2
  let table2 a b expr =
    [(true,  true,  eval2 a true  b true  expr);
     (true,  false, eval2 a true  b false expr);
     (false, true,  eval2 a false b true  expr);
     (false, false, eval2 a false b false expr) ];;
val eval2 : string -> bool -> string -> bool -> bool_expr -> bool = <fun>
val table2 : string -> string -> bool_expr -> (bool * bool * bool) list =
  <fun>

# table2 "a" "b" (And(Var "a", Or(Var "a", Var "b")));;
- : (bool * bool * bool) list =
[(true, true, true); (true, false, true); (false, true, false);
 (false, false, false)]
```

Truth tables for logical expressions. (*medium*)

Generalize the previous problem in such a way that the logical expression may contain any number of logical variables. Define `table` in a way that `table variables expr` returns the truth table for the expression `expr`, which contains the logical variables enumerated in `variables`.

Solution

```
# (* [val_vars] is an associative list containing the truth value of
     each variable.  For efficiency, a Map or a Hashtlb should be
     preferred. *)

  let rec eval val_vars = function
    | Var x -> List.assoc x val_vars
    | Not e -> not(eval val_vars e)
    | And(e1, e2) -> eval val_vars e1 && eval val_vars e2
    | Or(e1, e2) -> eval val_vars e1 || eval val_vars e2

  (* Again, this is an easy and short implementation rather than an
     efficient one. *)
  let rec table_make val_vars vars expr =
    match vars with
    | [] -> [(List.rev val_vars, eval val_vars expr)]
    | v :: tl ->
```

```
        table_make ((v, true) :: val_vars) tl expr
        @ table_make ((v, false) :: val_vars) tl expr

  let table vars expr = table_make [] vars expr;;
val eval : (string * bool) list -> bool_expr -> bool = <fun>
val table_make :
  (string * bool) list ->
  string list -> bool_expr -> ((string * bool) list * bool) list = <fun>
val table : string list -> bool_expr -> ((string * bool) list * bool) list =
  <fun>

# table ["a"; "b"] (And(Var "a", Or(Var "a", Var "b")));;
- : ((string * bool) list * bool) list =
[([("a", true); ("b", true)], true); ([("a", true); ("b", false)], true);
 ([("a", false); ("b", true)], false); ([("a", false); ("b", false)], false)]
# let a = Var "a" and b = Var "b" and c = Var "c" in
  table ["a"; "b"; "c"] (Or(And(a, Or(b,c)), Or(And(a,b), And(a,c))));;
- : ((string * bool) list * bool) list =
[([("a", true); ("b", true); ("c", true)], true);
 ([("a", true); ("b", true); ("c", false)], true);
 ([("a", true); ("b", false); ("c", true)], true);
 ([("a", true); ("b", false); ("c", false)], false);
 ([("a", false); ("b", true); ("c", true)], false);
 ([("a", false); ("b", true); ("c", false)], false);
 ([("a", false); ("b", false); ("c", true)], false);
 ([("a", false); ("b", false); ("c", false)], false)]
```

Gray code. (*medium*)

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.

Solution

```
# let prepend c s =
    (* Prepend the char [c] to the string [s]. *)
    let s' = String.create (String.length s + 1) in
    s'.[0] <- c;
    String.blit s 0 s' 1 (String.length s);
    s'

  let rec gray n =
    if n <= 1 then ["0"; "1"]
    else let g = gray (n - 1) in
         List.map (prepend '0') g @ List.rev_map (prepend '1') g;;
```

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

# gray 1;;
- : string list = ["0"; "1"]
# gray 2;;
- : string list = ["00"; "01"; "11"; "10"]
# gray 3;;
- : string list = ["000"; "001"; "011"; "010"; "110"; "111"; "101"; "100"]
```

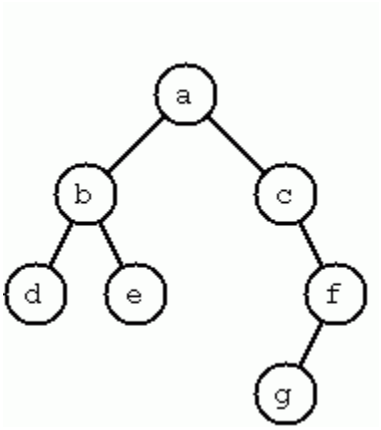
Huffman code. (*hard*)

First of all, consult a good book on discrete mathematics or algorithms for a detailed description of Huffman codes (you can start with the Wikipedia page)!

We suppose a set of symbols with their frequencies, given as a list of `Fr(S,F)` terms. Example: `fs = [Fr(a,45); Fr(b,13); Fr(c,12); Fr(d,16); Fr(e,9); Fr(f,5)]`. Our objective is to construct a list `Hc(S,C)` terms, where `C` is the Huffman code word for the symbol `S`. In our example, the result could be `hs = [Hc(a,'0'); Hc(b,'101'); Hc(c,'100'); Hc(d,'111'); Hc(e,'1101'); hc(f,'1100')]` [hc(a,'01'),...etc.]. The task shall be performed by the function `huffman` defined as follows: `huffman(fs)` returns the Huffman code table for the frequency table `fs`

```
(* example is pending *)
```

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` 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
```

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`.

Construct completely balanced binary trees. (*medium*)

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.

Solution

```
# (* Build all trees with given [left] and [right] subtrees. *)
  let add_trees_with left right all =
    let add_right_tree all l =
      List.fold_left (fun a r -> Node('x', l, r) :: a) all right in
    List.fold_left add_right_tree all left

  let rec cbal_tree n =
    if n = 0 then [Empty]
    else if n mod 2 = 1 then
      let t = cbal_tree (n / 2) in
      add_trees_with t t []
    else (* n even: n-1 nodes for the left & right subtrees altogether. *)
      let t1 = cbal_tree (n / 2 - 1) in
      let t2 = cbal_tree (n / 2) in
      add_trees_with t1 t2 (add_trees_with t2 t1 []);;
val add_trees_with :
  char binary_tree list ->
  char binary_tree list -> char binary_tree list -> char binary_tree list =
  <fun>
val cbal_tree : int -> char binary_tree list = <fun>

# cbal_tree 4;;
- : char binary_tree list =
[Node ('x', Node ('x', Empty, Empty),
  Node ('x', Node ('x', Empty, Empty), Empty));
 Node ('x', Node ('x', Empty, Empty),
  Node ('x', Empty, Node ('x', Empty, Empty)));
 Node ('x', Node ('x', Node ('x', Empty, Empty), Empty),
  Node ('x', Empty, Empty));
 Node ('x', Node ('x', Empty, Node ('x', Empty, Empty)),
  Node ('x', Empty, Empty))]
# List.length(cbal_tree 40);;
- : int = 524288
```

Symmetric binary trees. (*medium*)

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.

Solution

```
# let rec is_mirror t1 t2 =
    match t1, t2 with
    | Empty, Empty -> true
    | Node(_, l1, r1), Node(_, l2, r2) ->
       is_mirror l1 r2 && is_mirror r1 l2
    | _ -> false

  let is_symmetric = function
    | Empty -> true
    | Node(_, l, r) -> is_mirror l r;;
val is_mirror : 'a binary_tree -> 'b binary_tree -> bool = <fun>
val is_symmetric : 'a binary_tree -> bool = <fun>
```

Binary search trees (dictionaries). (*medium*)

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

Solution

```
# let rec insert tree x = match tree with
    | Empty -> Node(x, Empty, Empty)
    | Node(y, l, r) ->
       if x = y then tree
       else if x < y then Node(y, insert l x, r)
       else Node(y, l, insert r x)
  let construct l = List.fold_left insert Empty l;;
val insert : 'a binary_tree -> 'a -> 'a binary_tree = <fun>
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
```

Generate-and-test paradigm. (*medium*)

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

Solution

```
# let sym_cbal_trees n =
    List.filter is_symmetric (cbal_tree n);;
val sym_cbal_trees : int -> char binary_tree list = <fun>

# sym_cbal_trees 5;;
- : char binary_tree list =
[Node ('x', Node ('x', Node ('x', Empty, Empty), Empty),
  Node ('x', Empty, Node ('x', Empty, Empty)));
 Node ('x', Node ('x', Empty, Node ('x', Empty, Empty)),
  Node ('x', Node ('x', Empty, Empty), Empty))]
```

How many such trees are there with 57 nodes? Investigate about how many solutions there are for a given number of nodes? What if the number is even? Write an appropriate function.

```
# List.length (sym_cbal_trees 57);;
- : int = 256
# List.map (fun n -> n, List.length(sym_cbal_trees n)) (range 10 20);;
- : (int * int) list =
[(10, 0); (11, 4); (12, 0); (13, 4); (14, 0); (15, 1); (16, 0); (17, 8);
 (18, 0); (19, 16); (20, 0)]
```

Construct height-balanced binary trees. (*medium*)

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.

Solution

The function `add_trees_with` is defined in the solution of [Construct completely balanced binary trees](#Constructcompletelybalancedbinarytreesmedium).

```
# let rec hbal_tree n =
```

```
    if n = 0 then [Empty]
    else if n = 1 then [Node('x', Empty, Empty)]
    else
    (* [add_trees_with left right trees] is defined in a question above. *)
      let t1 = hbal_tree (n - 1)
      and t2 = hbal_tree (n - 2) in
      add_trees_with t1 t1 (add_trees_with t1 t2 (add_trees_with t2 t1 []));;
val hbal_tree : int -> char binary_tree list = <fun>

# let t = hbal_tree 3;;
val t : char binary_tree list =
  [Node ('x', Node ('x', Empty, Node ('x', Empty, Empty)),
    Node ('x', Empty, Node ('x', Empty, Empty)));
   Node ('x', Node ('x', Empty, Node ('x', Empty, Empty)),
    Node ('x', Node ('x', Empty, Empty), Empty));
   Node ('x', Node ('x', Empty, Node ('x', Empty, Empty)),
    Node ('x', Node ('x', Empty, Empty), Node ('x', Empty, Empty)));
   Node ('x', Node ('x', Node ('x', Empty, Empty), Empty),
    Node ('x', Empty, Node ('x', Empty, Empty)));
   Node ('x', Node ('x', Node ('x', Empty, Empty), Empty),
    Node ('x', Node ('x', Empty, Empty), Empty));
   Node ('x', Node ('x', Node ('x', Empty, Empty), Empty),
    Node ('x', Node ('x', Empty, Empty), Node ('x', Empty, Empty)));
   Node ('x', Node ('x', Node ('x', Empty, Empty), Node ('x', Empty, Empty)),
    Node ('x', Empty, Node ('x', Empty, Empty)));
   Node ('x', Node ('x', Node ('x', Empty, Empty), Node ('x', Empty, Empty)),
    Node ('x', Node ('x', Empty, Empty), Empty));
   Node ('x', Node ('x', Node ('x', Empty, Empty), Node ('x', Empty, Empty)),
    Node ('x', Node ('x', Empty, Empty), Node ('x', Empty, Empty)));
   Node ('x', Node ('x', Empty, Node ('x', Empty, Empty)),
    Node ('x', Empty, Empty));
   Node ('x', Node ('x', Node ('x', Empty, Empty), Empty),
    Node ('x', Empty, Empty));
   Node ('x', Node ('x', Node ('x', Empty, Empty), Node ('x', Empty, Empty)),
    Node ('x', Empty, Empty));
   Node ('x', Node ('x', Empty, Empty),
    Node ('x', Empty, Node ('x', Empty, Empty)));
   Node ('x', Node ('x', Empty, Empty),
    Node ('x', Node ('x', Empty, Empty), Empty));
   Node ('x', Node ('x', Empty, Empty),
    Node ('x', Node ('x', Empty, Empty), Node ('x', Empty, Empty)))]
# let x = 'x';;
val x : char = 'x'
# List.mem (Node(x, Node(x, Node(x, Empty, Empty), Node(x, Empty, Empty)),
                 Node(x, Node(x, Empty, Empty), Node(x, Empty, Empty)))) t;;
- : bool = true
# List.mem (Node(x, Node(x, Node(x, Empty, Empty), Node(x, Empty, Empty)),
                 Node(x, Node(x, Empty, Empty), Empty))) t;;
- : bool = true
# List.length t;;
- : int = 15
```

Construct height-balanced binary trees with a given number of nodes. (*medium*)

Consider a height-balanced binary tree of height `h`. What is the maximum number of nodes it can contain? Clearly, $maxN = 2^h - 1$. However, what is the minimum number *minN*? This question is more difficult. Try to find a recursive statement and turn it into a function `min_nodes` defined as follows: `min_nodes h` returns the minimum number of nodes in a height-balanced binary tree of height `h`.

Solution

```
# let rec min_nodes h =
    if h <= 0 then 0
    else if h = 1 then 1
    else min_nodes (h - 1) + min_nodes (h - 2) + 1;;
val min_nodes : int -> int = <fun>
```

On the other hand, we might ask: what is the maximum height H a height-balanced binary tree with N nodes can have? `max_height n` returns the maximum height of a height-balanced binary tree with `n` nodes.

Solution

```
# let rec max_height = function
    | 0 -> 0
    | n ->
      let h = max_height (n - 1) in
      if max_height (n - min_nodes (h - 1) - 1) = h then h + 1 else h;;
val max_height : int -> int = <fun>
```

Now, we can attack the main problem: construct all the height-balanced binary trees with a given nuber of nodes. `hbal_tree_nodes n` returns a list of all height-balanced binary tree with `n` nodes.

Find out how many height-balanced trees exist for `n = 15`.

```
# List.length (hbal_tree_nodes 15);;
Error: Unbound value hbal_tree_nodes
```

Count the leaves of a binary tree. (*easy*)

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

Solution

```
# let rec count_leaves = function
    | Empty -> 0
    | Node(_, Empty, Empty) -> 1
    | Node(_, l, r) -> count_leaves l + count_leaves r;;
```

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

```
# count_leaves Empty;;
- : int = 0
# count_leaves example_tree;;
- : int = 3
```

Collect the leaves of a binary tree in a list. (*easy*)

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

Solution

```
# let rec leaves = function
    | Empty -> []
    | Node(c, Empty, Empty) -> [c]
    | Node(_, l, r) -> leaves l @ leaves r;;
val leaves : 'a binary_tree -> 'a list = <fun>
```

```
# leaves Empty;;
- : 'a list = []
# leaves example_tree;;
- : char list = ['d'; 'e'; 'g']
```

Collect the internal nodes of a binary tree in a list. (*easy*)

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.

Solution

```
# let rec internals = function
    | Empty | Node(_, Empty, Empty) -> []
    | Node(c, l, r) -> internals l @ (c :: internals r);;
val internals : 'a binary_tree -> 'a list = <fun>
```

```
# internals (Node('a', Empty, Empty));;
- : char list = []
# internals example_tree;;
- : char list = ['b'; 'a'; 'c'; 'f']
```

Collect the nodes at a given level in a list. (*easy*)

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.

Solution

```
# let rec at_level t l = match t with
    | Empty -> []
    | Node(c, left, right) ->
       if l = 1 then [c]
       else at_level left (l - 1) @ at_level right (l - 1);;
val at_level : 'a binary_tree -> int -> 'a list = <fun>

# at_level example_tree 2;;
- : char list = ['b'; 'c']
# at_level example_tree 5;;
- : char list = []
```

Using `at_level` it is easy to construct a function `levelorder` which creates the level-order sequence of the nodes. However, there are more efficient ways to do that.

Construct a complete binary tree. (*medium*)

A *complete* binary tree with height H is defined as follows: The levels 1,2,3,...,H-1 contain the maximum number of nodes (i.e 2^{i-1} at the level i, note that we start counting the levels from 1 at the root). In level H, which may contain less than the maximum possible number of nodes, all the nodes are "left-adjusted". This means that in a levelorder tree traversal all internal nodes come first, the leaves come second, and empty successors (the nil's which are not really nodes!) come last.

Particularly, complete binary trees are used as data structures (or addressing schemes) for heaps.

We can assign an address number to each node in a complete binary tree by enumerating the nodes in levelorder, starting at the root with number 1. In doing so, we realize that for every node X with address A the following property holds: The address of X's left and right successors are 2*A and 2*A+1, respectively, supposed the successors do exist. This fact can be used to elegantly construct a complete binary tree structure. Write a function `is_complete_binary_tree` with the following specification: `is_complete_binary_tree n t` returns `true` iff `t` is a complete binary tree with `n` nodes.

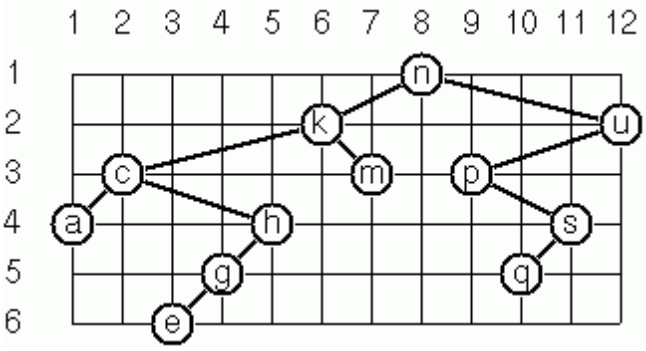
Solution

```
# let rec split_n lst acc n = match (n, lst) with
    | (0, _) -> (List.rev acc, lst)
    | (_, []) -> (List.rev acc, [])
    | (_, h::t) -> split_n t (h::acc) (n-1)
```

```
  let rec myflatten p c =
    match (p, c) with
    | (p, []) -> List.map (fun x -> Node (x, Empty, Empty)) p
    | (x::t, [y]) -> Node (x, y, Empty)::myflatten t []
    | (ph::pt, x::y::t) -> (Node (ph, x, y))::(myflatten pt t)
    | _ -> invalid_arg "myflatten"

  let complete_binary_tree = function
    | [] -> Empty
    | lst ->
       let rec aux l = function
         | [] -> []
         | lst -> let p, c = split_n lst [] (1 lsl l) in
                  myflatten p (aux (l+1) c) in
       List.hd (aux 0 lst);;
val split_n : 'a list -> 'a list -> int -> 'a list * 'a list = <fun>
val myflatten : 'a list -> 'a binary_tree list -> 'a binary_tree list = <fun>
val complete_binary_tree : 'a list -> 'a binary_tree = <fun>

# complete_binary_tree [1;2;3;4;5;6];;
- : int binary_tree =
Node (1, Node (2, Node (4, Empty, Empty), Node (5, Empty, Empty)),
 Node (3, Node (6, Empty, Empty), Empty))
```

Layout a binary tree (1). (*medium*)

As a preparation for drawing the tree, a layout algorithm is required to determine the position of each node in a rectangular grid. Several layout methods are conceivable, one of them is shown in the illustration.



In this layout strategy, the position of a node v is obtained by the following two rules:

- *x(v)* is equal to the position of the node v in the *inorder* sequence;
- *y(v)* is equal to the depth of the node *v* in the tree.

In order to store the position of the nodes, we redefine the OCaml type representing a node (and its successors) as follows:

```
# type 'a pos_binary_tree =
    | E (* represents the empty tree *)
    | N of 'a * int * int * 'a pos_binary_tree * 'a pos_binary_tree;;
type 'a pos_binary_tree =
    E
  | N of 'a * int * int * 'a pos_binary_tree * 'a pos_binary_tree
```

`N(w,x,y,l,r)` represents a (non-empty) binary tree with root `w` "positioned" at `(x,y)`,

and subtrees `l` and `r`. Write a function `layout_binary_tree_1` with the following specification: `layout_binary_tree_1 t` returns the "positioned" binary tree obtained from the binary tree `t`.

The tree pictured above is

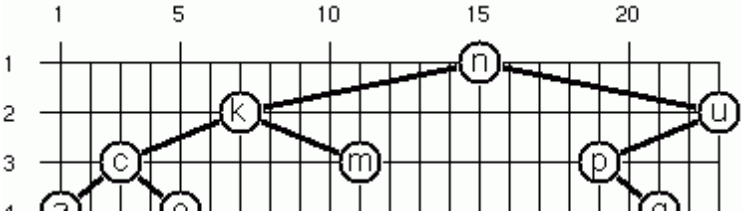
```
# let example_layout_tree =
    let leaf x = Node (x, Empty, Empty) in
    Node('n', Node('k', Node('c', leaf 'a',
                             Node('h', Node('g', leaf 'e',Empty), Empty)),
                    leaf 'm'),
          Node('u', Node('p', Empty, Node('s', leaf 'q', Empty)), Empty));;
val example_layout_tree : char binary_tree =
  Node ('n',
   Node ('k',
    Node ('c', Node ('a', Empty, Empty),
     Node ('h', Node ('g', Node ('e', Empty, Empty), Empty), Empty)),
    Node ('m', Empty, Empty)),
   Node ('u', Node ('p', Empty, Node ('s', Node ('q', Empty, Empty), Empty)),
    Empty))
```

Solution

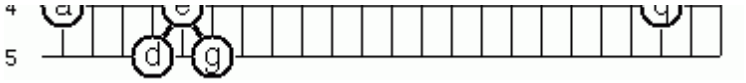
```
# let layout_binary_tree_1 t =
    let rec layout depth x_left = function
      (* This function returns a pair: the laid out tree and the first
       * free x location *)
      | Empty -> (E,x_left)
      | Node (x,l,r) ->
         let (l',l_x_max) = layout (depth + 1) x_left l in
         let (r',r_x_max) = layout (depth + 1) (l_x_max + 1) r in
         (N (x,l_x_max,depth,l',r'),r_x_max)
    in fst (layout 1 1 t);;
val layout_binary_tree_1 : 'a binary_tree -> 'a pos_binary_tree = <fun>

# layout_binary_tree_1 example_layout_tree;;
- : char pos_binary_tree =
N ('n', 8, 1,
 N ('k', 6, 2,
  N ('c', 2, 3, N ('a', 1, 4, E, E),
   N ('h', 5, 4, N ('g', 4, 5, N ('e', 3, 6, E, E), E), E)),
  N ('m', 7, 3, E, E)),
 N ('u', 12, 2, N ('p', 9, 3, E, N ('s', 11, 4, N ('q', 10, 5, E, E), E)), E))
```

Layout a binary tree (2). (*medium*)

An alternative layout method is depicted in this illustration. Find out the rules and write the corresponding OCaml function.



Hint: On a given level, the horizontal distance between neighbouring nodes is constant.



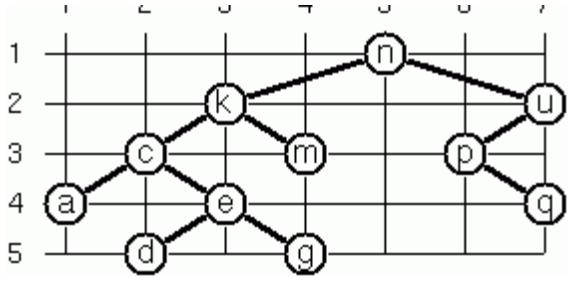
The tree shown is

```
# let example_layout_tree =
    let leaf x = Node (x,Empty,Empty) in
    Node('n', Node('k', Node('c', leaf 'a',
                             Node('e', leaf 'd', leaf 'g')),
                    leaf 'm'),
          Node('u', Node('p', Empty, leaf 'q'), Empty));;
val example_layout_tree : char binary_tree =
  Node ('n',
   Node ('k',
    Node ('c', Node ('a', Empty, Empty),
     Node ('e', Node ('d', Empty, Empty), Node ('g', Empty, Empty))),
    Node ('m', Empty, Empty)),
   Node ('u', Node ('p', Empty, Node ('q', Empty, Empty)), Empty))
```

Solution

```
# let layout_binary_tree_2 t =
    let rec height = function
      | Empty -> 0
      | Node (_,l,r) -> 1 + max (height l) (height r) in
    let tree_height = height t in
    let rec find_missing_left depth = function
      | Empty -> tree_height - depth
      | Node (_,l,_) -> find_missing_left (depth + 1) l in
    let translate_dst = find_missing_left 0 t in
    let rec layout depth x_root = function
      | Empty -> E
      | Node (x,l,r) ->
         let spacing = 1 lsl (tree_height - depth - 1) in
         let l' = layout (depth + 1) (x_root - spacing) l
         and r' = layout (depth + 1) (x_root + spacing) r in
         N (x, x_root,depth, l',r') in
    layout 1 ((1 lsl (tree_height - 1)) - translate_dst) t;;
val layout_binary_tree_2 : 'a binary_tree -> 'a pos_binary_tree = <fun>

# layout_binary_tree_2 example_layout_tree;;
- : char pos_binary_tree =
N ('n', 15, 1,
 N ('k', 7, 2,
  N ('c', 3, 3, N ('a', 1, 4, E, E),
   N ('e', 5, 4, N ('d', 4, 5, E, E), N ('g', 6, 5, E, E))),
  N ('m', 11, 3, E, E)),
 N ('u', 23, 2, N ('p', 19, 3, E, N ('q', 21, 4, E, E)), E))
```

Layout a binary tree (3). (*hard*)



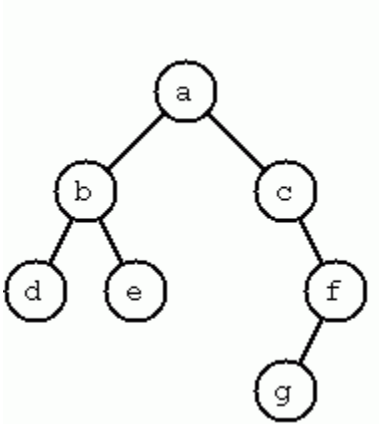
Yet another layout strategy is shown in the above illustration. The method yields a very compact layout while maintaining a certain symmetry in every node. Find out the rules and write the corresponding predicate.

Hint: Consider the horizontal distance between a node and its successor nodes. How tight can you pack together two subtrees to construct the combined binary tree? This is a difficult problem. Don't give up too early!

```
(* solution pending *)
```

Which layout do you like most?

A string representation of binary trees. (*medium*)



Somebody represents binary trees as strings of the following type (see example): `"a(b(d,e),c(,f(g,)))"`.

- Write an OCaml function which generates this string representation, if the tree is given as usual (as `Empty` or `Node(x,l,r)` term). Then write a function which does this inverse; i.e. given the string representation, construct the tree in the usual form. Finally, combine the two predicates in a single function `tree_string` which can be used in both directions.
- Write the same predicate `tree_string` using difference lists and a single predicate `tree_dlist` which does the conversion between a tree and a difference list in both directions.

For simplicity, suppose the information in the nodes is a single letter and there are no spaces in the string.

```
(* solution pending *)
```

Preorder and inorder sequences of binary trees. (*medium*)

We consider binary trees with nodes that are identified by single lower-case letters, as in the example of the previous problem.

1. Write functions `preorder` and `inorder` that construct the preorder and inorder sequence of a given binary tree, respectively. The results should be atoms, e.g. 'abdecfg' for the preorder sequence of the example in the previous

problem.
2. Can you use `preorder` from problem part 1 in the reverse direction; i.e. given a preorder sequence, construct a corresponding tree? If not, make the necessary arrangements.
3. If both the preorder sequence and the inorder sequence of the nodes of a binary tree are given, then the tree is determined unambiguously. Write a function `pre_in_tree` that does the job.
4. Solve problems 1 to 3 using difference lists. Cool! Use the function `timeit` (defined in problem "Compare the two methods of calculating Euler's totient function.") to compare the solutions.

What happens if the same character appears in more than one node. Try for instance `pre_in_tree "aba" "baa"`.

Solution

We use lists to represent the result. Note that `preorder` and `inorder` can be made more efficient by avoiding list concatenations.

```
# let rec preorder = function
    | Empty -> []
    | Node (v, l, r) -> v :: (preorder l @ preorder r)

  let rec inorder = function
    | Empty -> []
    | Node (v, l, r) -> inorder l @ (v :: inorder r)

  let rec split_pre_in p i x accp acci = match (p, i) with
    | [], [] -> (List.rev accp, List.rev acci), ([], [])
    | h1::t1, h2::t2 ->
       if x=h2 then
         (List.tl (List.rev (h1::accp)), t1),
         (List.rev (List.tl (h2::acci)), t2)
       else
         split_pre_in t1 t2 x (h1::accp) (h2::acci)
    | _ -> assert false

  let rec pre_in_tree p i = match (p, i) with
    | [], [] -> Empty
    | (h1::t1), (h2::t2) ->
       let (lp, rp), (li, ri) = split_pre_in p i h1 [] [] in
       Node (h1, pre_in_tree lp li, pre_in_tree rp ri)
    | _ -> invalid_arg "pre_in_tree";;
val preorder : 'a binary_tree -> 'a list = <fun>
val inorder : 'a binary_tree -> 'a list = <fun>
val split_pre_in :
  'a list ->
  'a list ->
  'a -> 'a list -> 'a list -> ('a list * 'a list) * ('a list * 'a list) =
  <fun>
val pre_in_tree : 'a list -> 'a list -> 'a binary_tree = <fun>
```

```
# preorder (Node (1, Node (2, Empty, Empty), Empty));;
- : int list = [1; 2]
# preorder (Node (1, Empty, Node (2, Empty, Empty)));;
- : int list = [1; 2]
# let p = preorder example_tree;;
val p : char list = ['a'; 'b'; 'd'; 'e'; 'c'; 'f'; 'g']
# let i = inorder example_tree;;
val i : char list = ['d'; 'b'; 'e'; 'a'; 'c'; 'g'; 'f']
# pre_in_tree p i = example_tree;;
- : bool = true
```

Solution using difference lists.

```
  (* solution pending *)
```

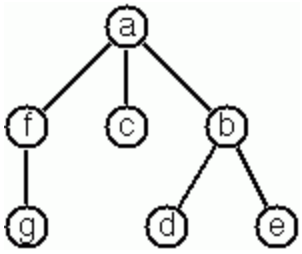
Dotstring representation of binary trees. (*medium*)

We consider again binary trees with nodes that are identified by single lower-case letters, as in the example of problem "A string representation of binary trees". Such a tree can be represented by the preorder sequence of its nodes in which dots (.) are inserted where an empty subtree (nil) is encountered during the tree traversal. For example, the tree shown in problem "A string representation of binary trees" is represented as 'abd..e..c.fg...'. First, try to establish a syntax (BNF or syntax diagrams) and then write a function `tree_dotstring` which does the conversion in both directions. Use difference lists.

```
  (* solution pending *)
```

Multiway Trees

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

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

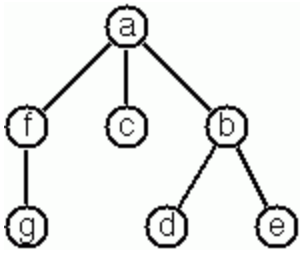
```
# T('a', [T('f',[T('g',[])]); T('c',[]); T('b',[T('d',[]); T('e',[])])]);;
- : char mult_tree =
T ('a',
 [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e', [])])])
```

Count the nodes of a multiway tree. (*easy*)

Solution

```
# let rec count_nodes (T(_, sub)) =
    List.fold_left (fun n t -> n + count_nodes t) 1 sub;;
val count_nodes : 'a mult_tree -> int = <fun>

# count_nodes (T('a', [T('f',[]) ]));;
- : int = 2
```

Tree construction from a node string. (*medium*)



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.

Solution

```
# (* We could build the final string by string concatenation but
     this is expensive due to the number of operations.  We use a
     buffer instead. *)
  let rec add_string_of_tree buf (T(c, sub)) =
    Buffer.add_char buf c;
    List.iter (add_string_of_tree buf) sub;
    Buffer.add_char buf '^'
  let string_of_tree t =
    let buf = Buffer.create 128 in
    add_string_of_tree buf t;
    Buffer.contents buf;;
val add_string_of_tree : Buffer.t -> char mult_tree -> unit = <fun>
val string_of_tree : char mult_tree -> string = <fun>
# let rec tree_of_substring t s i len =
    if i >= len || s.[i] = '^' then List.rev t, i + 1
    else
      let sub, j = tree_of_substring [] s (i+1) len in
      tree_of_substring (T(s.[i], sub) ::t) s j len
  let tree_of_string s =
    match tree_of_substring [] s 0 (String.length s) with
    | [t], _ -> t
    | _ -> failwith "tree_of_string";;
```

```
val tree_of_substring :
  char mult_tree list -> string -> int -> int -> char mult_tree list * int =
  <fun>
val tree_of_string : string -> char mult_tree = <fun>

# let t = T('a', [T('f',[T('g',[])]); T('c',[]);
            T('b',[T('d',[]); T('e',[])])]);;
val t : char mult_tree =
  T ('a',
   [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e', [])])])
# string_of_tree t;;
- : string = "afg^^c^bd^e^^^"
# tree_of_string "afg^^c^bd^e^^^";;
- : char mult_tree =
T ('a',
 [T ('f', [T ('g', [])]); T ('c', []); T ('b', [T ('d', []); T ('e', [])])])
```

Determine the internal path length of a tree. (*easy*)

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`.

Solution

```
# let rec ipl_sub len (T(_, sub)) =
    (* [len] is the distance of the current node to the root.  Add the
       distance of all sub-nodes. *)
    List.fold_left (fun sum t -> sum + ipl_sub (len + 1) t) len sub
  let ipl t = ipl_sub 0 t;;
val ipl_sub : int -> 'a mult_tree -> int = <fun>
val ipl : 'a mult_tree -> int = <fun>

# ipl t;;
- : int = 9
```

Construct the bottom-up order sequence of the tree nodes. (*easy*)

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

Solution

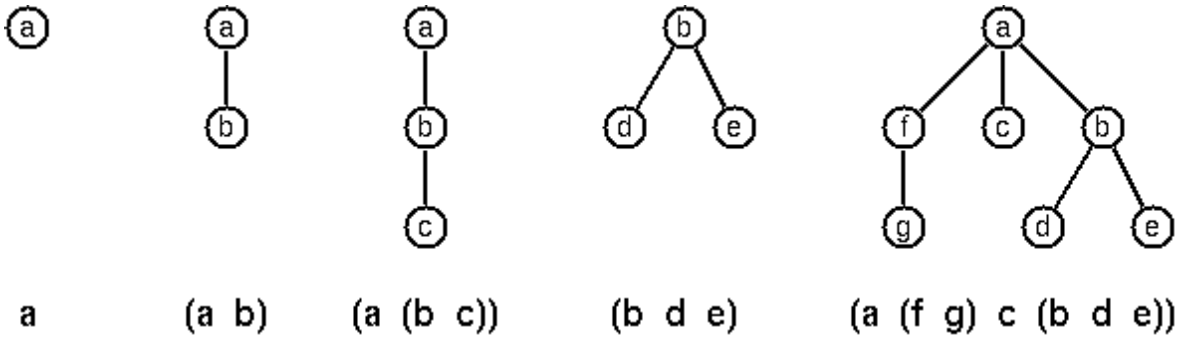
```
# let rec prepend_bottom_up (T(c, sub)) l =
    List.fold_right (fun t l -> prepend_bottom_up t l) sub (c :: l)
  let bottom_up t = prepend_bottom_up t [];;
val prepend_bottom_up : 'a mult_tree -> 'a list -> 'a list = <fun>
val bottom_up : 'a mult_tree -> 'a list = <fun>
```

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

Lisp-like tree representation. (*medium*)

There is a particular notation for multiway trees in Lisp. The picture shows how multiway tree structures are represented in Lisp.

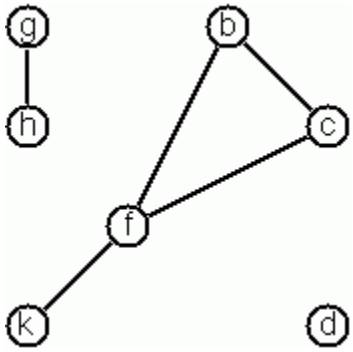
Note that in the "lispy" notation a node with successors (children) in the tree is always the first element in a list, followed by its children. The "lispy" representation of a multiway tree is a sequence of atoms and parentheses '(' and ')'. This is very close to the way trees are represented in OCaml, except that no constructor `T` is used. Write a function `lispy : char mult_tree -> string` that returns the lispy notation of the tree.



Solution

```
# let rec add_lispy buf = function
    | T(c, []) -> Buffer.add_char buf c
    | T(c, sub) ->
       Buffer.add_char buf '(';
       Buffer.add_char buf c;
       List.iter (fun t -> Buffer.add_char buf ' '; add_lispy buf t) sub;
       Buffer.add_char buf ')'
  let lispy t =
    let buf = Buffer.create 128 in
    add_lispy buf t;
    Buffer.contents buf;;
val add_lispy : Buffer.t -> char mult_tree -> unit = <fun>
val lispy : char mult_tree -> string = <fun>

# lispy (T('a', []));;
- : string = "a"
# lispy (T('a', [T('b', [])]));;
- : string = "(a b)"
# lispy t;;
- : string = "(a (f g) c (b d e))"
```

Graphs



A graph is defined as a set of nodes and a set of edges, where each edge is a pair of different nodes.

There are several ways to represent graphs in OCaml.

- One method is to list all edges, an edge being a pair of nodes. In this form, the graph depicted opposite is represented as the following expression:

```
# ['h', 'g';  'k', 'f';  'f', 'b';  'f', 'c';  'c', 'b'];;
- : (char * char) list =
[('h', 'g'); ('k', 'f'); ('f', 'b'); ('f', 'c'); ('c', 'b')]
```

We call this **edge-clause form**. Obviously, isolated nodes cannot be represented.

- Another method is to represent the whole graph as one data object. According to the definition of the graph as a pair of two sets (nodes and edges), we may use the following OCaml type:

```
# type 'a graph_term = { nodes : 'a list;  edges : ('a * 'a) list };;
type 'a graph_term = { nodes : 'a list; edges : ('a * 'a) list; }
```

Then, the above example graph is represented by:

```
# let example_graph =
    { nodes = ['b'; 'c'; 'd'; 'f'; 'g'; 'h'; 'k'];
      edges = ['h', 'g';  'k', 'f';  'f', 'b';  'f', 'c';  'c', 'b'] };;
val example_graph : char graph_term =
  {nodes = ['b'; 'c'; 'd'; 'f'; 'g'; 'h'; 'k'];
   edges = [('h', 'g'); ('k', 'f'); ('f', 'b'); ('f', 'c'); ('c', 'b')]}
```

We call this **graph-term form**. Note, that the lists are kept sorted, they are really sets, without duplicated elements. Each edge appears only once in the edge list; i.e. an edge from a node x to another node y is represented as `(x,y)`, the couple `(y,x)` is not present. The **graph-term form is our default representation.** You may want to define a similar type using sets instead of lists.

- A third representation method is to associate with each node the set of nodes that are adjacent to that node. We call this the **adjacency-list form**. In our example:

  ```
  (* example pending *)
  ```

- The representations we introduced so far well suited for automated

processing, but their syntax is not very user-friendly. Typing the terms by hand is cumbersome and error-prone. We can define a more compact and "human-friendly" notation as follows: A graph (with char labelled nodes) is represented by a string of atoms and terms of the type X-Y. The atoms stand for isolated nodes, the X-Y terms describe edges. If an X appears as an endpoint of an edge, it is automatically defined as a node. Our example could be written as:

```
"b-c f-c g-h d f-b k-f h-g"
```

We call this the **human-friendly form**. As the example shows, the list does not have to be sorted and may even contain the same edge multiple times. Notice the isolated node `d`.

Conversions. (*easy*)

Write functions to convert between the different graph representations. With these functions, all representations are equivalent; i.e. for the following problems you can always pick freely the most convenient form. This problem is not particularly difficult, but it's a lot of work to deal with all the special cases.

```
(* example pending *)
```

Path from one node to another one. (*medium*)

Write a function `paths g a b` that returns all acyclic path `p` from node `a` to node `b` ≠ `a` in the graph `g`. The function should return the list of all paths via backtracking.

Solution

```
# (* The datastructures used here are far from the most efficient ones
     but allow for a straightforward implementation. *)
  (* Returns all neighbors satisfying the condition. *)
  let neighbors g a cond =
    let edge l (b,c) = if b = a && cond c then c :: l
                       else if c = a && cond b then b :: l
                       else l in
    List.fold_left edge [] g.edges
  let rec list_path g a to_b = match to_b with
    | [] -> assert false (* [to_b] contains the path to [b]. *)
    | a' :: _ ->
       if a' = a then [to_b]
       else
         let n = neighbors g a' (fun c -> not(List.mem c to_b)) in
         List.concat(List.map (fun c -> list_path g a (c :: to_b)) n)

  let paths g a b =
    assert(a <> b);
```

```
    list_path g a [b];;
val neighbors : 'a graph_term -> 'a -> ('a -> bool) -> 'a list = <fun>
val list_path : 'a graph_term -> 'a -> 'a list -> 'a list list = <fun>
val paths : 'a graph_term -> 'a -> 'a -> 'a list list = <fun>

# paths example_graph 'f' 'b';;
- : char list list = [['f'; 'c'; 'b']; ['f'; 'b']]
```

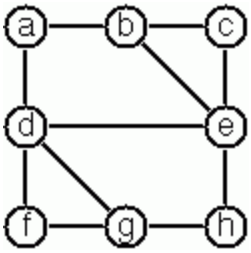
Cycle from a given node. (*easy*)

Write a functions `cycle g a` that returns a closed path (cycle) `p` starting at a given node `a` in the graph `g`. The predicate should return the list of all cycles via backtracking.

Solution

```
# let cycles g a =
    let n = neighbors g a (fun _ -> true) in
    let p = List.concat(List.map (fun c -> list_path g a [c]) n) in
    List.map (fun p -> p @ [a]) p;;
val cycles : 'a graph_term -> 'a -> 'a list list = <fun>

# cycles example_graph 'f';;
- : char list list =
[['f'; 'b'; 'c'; 'f']; ['f'; 'c'; 'f']; ['f'; 'c'; 'b'; 'f'];
 ['f'; 'b'; 'f']; ['f'; 'k'; 'f']]
```

Construct all spanning trees. (*medium*)

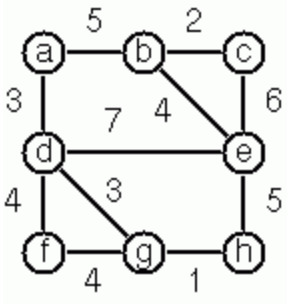


Write a function `s_tree g` to construct (by backtracking) all spanning trees of a given graph `g`. With this predicate, find out how many spanning trees there are for the graph depicted to the left. The data of this example graph can be found in the test below. When you have a correct solution for the `s_tree` function, use it to define two other useful functions: `is_tree graph` and `is_connected Graph`. Both are five-minutes tasks!

```
(* solution pending *);;

# let g = { nodes = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'];
            edges = [('a', 'b'); ('a', 'd'); ('b', 'c'); ('b', 'e');
                     ('c', 'e'); ('d', 'e'); ('d', 'f'); ('d', 'g');
                     ('e', 'h'); ('f', 'g'); ('g', 'h')] };;
val g : char graph_term =
  {nodes = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'];
   edges =
    [('a', 'b'); ('a', 'd'); ('b', 'c'); ('b', 'e'); ('c', 'e'); ('d', 'e');
     ('d', 'f'); ('d', 'g'); ('e', 'h'); ('f', 'g'); ('g', 'h')]}
```

Construct the minimal spanning tree. (*medium*)

Write a function `ms_tree graph` to construct the minimal spanning tree of a given labelled graph. A labelled graph will be represented as follows:




```
# type ('a, 'b) labeled_graph = { nodes : 'a list;
                                  edges : ('a * 'a * 'b) list };;
type ('a, 'b) labeled_graph = {
  nodes : 'a list;
  edges : ('a * 'a * 'b) list;
}
```

(Beware that from now on `nodes` and `edges` mask the previous fields of the same name.)

Hint: Use the algorithm of Prim. A small modification of the solution of P83 does the trick. The data of the example graph to the right can be found below.

```
(* solution pending *);;
```

```
# let g = { nodes = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'];
            edges = [('a', 'b', 5); ('a', 'd', 3); ('b', 'c', 2);
                     ('b', 'e', 4); ('c', 'e', 6); ('d', 'e', 7);
                     ('d', 'f', 4); ('d', 'g', 3); ('e', 'h', 5);
                     ('f', 'g', 4); ('g', 'h', 1)] };;
val g : (char, int) labeled_graph =
  {nodes = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'];
   edges =
    [('a', 'b', 5); ('a', 'd', 3); ('b', 'c', 2); ('b', 'e', 4);
     ('c', 'e', 6); ('d', 'e', 7); ('d', 'f', 4); ('d', 'g', 3);
     ('e', 'h', 5); ('f', 'g', 4); ('g', 'h', 1)]}
```

Graph isomorphism. (*medium*)

Two graphs G1(N1,E1) and G2(N2,E2) are isomorphic if there is a bijection f: N1 → N2 such that for any nodes X,Y of N1, X and Y are adjacent if and only if f(X) and f(Y) are adjacent.

Write a function that determines whether two graphs are isomorphic. Hint: Use an open-ended list to represent the function f.

```
(* example pending *);;
```

Node degree and graph coloration. (*medium*)

- Write a function `degree graph node` that determines the degree of a given node.
- Write a function that generates a list of all nodes of a graph sorted according to decreasing degree.
- Use Welsh-Powell's algorithm to paint the nodes of a graph in such a way

that adjacent nodes have different colors.

```
(* example pending *);;
```

Depth-first order graph traversal. (*medium*)

Write a function that generates a depth-first order graph traversal sequence. The starting point should be specified, and the output should be a list of nodes that are reachable from this starting point (in depth-first order).

Specifically, the graph will be provided by its adjacency-list representation and you must create a module `M` with the following signature:

```
# module type GRAPH = sig
    type node = char
    type t
    val of_adjacency : (node * node list) list -> t
    val dfs_fold : t -> node -> ('a -> node -> 'a) -> 'a -> 'a
  end;;
module type GRAPH =
  sig
    type node = char
    type t
    val of_adjacency : (node * node list) list -> t
    val dfs_fold : t -> node -> ('a -> node -> 'a) -> 'a -> 'a
  end
```

where `M.dfs_fold g n f a` applies `f` on the nodes of the graph `g` in depth first order, starting with node `n`.

Solution

In a depth-first search you fully explore the edges of the most recently discovered node *v* before 'backtracking' to explore edges leaving the node from which *v* was discovered. To do a depth-first search means keeping careful track of what vertices have been visited and when. We compute timestamps for each vertex discovered in the search. A discovered vertex has two timestamps associated with it : its discovery time \(in map `d`\) and its finishing time \(in map `f`\) \(a vertex is finished when its adjacency list has been completely examined\). These timestamps are often useful in graph algorithms and aid in reasoning about the behavior of depth-first search. We color nodes during the search to help in the bookkeeping \(map `color`\). All vertices of the graph are initially `White`. When a vertex is discovered it is marked `Gray` and when it is finished, it is marked `Black`. If vertex *v* is discovered in the adjacency list of previously discovered node *u*, this fact is recorded in the predecessor subgraph \(map `pred`\).

```
# module M : GRAPH = struct
```

```
    module Char_map = Map.Make (Char)
    type node = char
    type t = (node list) Char_map.t

    let of_adjacency l =
      List.fold_right (fun (x, y) -> Char_map.add x y) l Char_map.empty

    type colors = White|Gray|Black

    type 'a state = {
      d : int Char_map.t ; (*discovery time*)
      f : int Char_map.t ; (*finishing time*)
      pred : char Char_map.t ; (*predecessor*)
      color : colors Char_map.t ; (*vertex colors*)
      acc : 'a ; (*user specified type used by 'fold'*)
    }

    let dfs_fold g c fn acc =
      let rec dfs_visit t u {d; f; pred; color; acc} =
        let edge (t, state) v =
          if Char_map.find v state.color = White then
            dfs_visit t v {state with pred=Char_map.add v u state.pred;}
          else  (t, state)
        in
        let t, {d; f; pred; color; acc} =
          let t = t + 1 in
          List.fold_left edge
            (t, {d=Char_map.add u t d; f;
                 pred; color=Char_map.add u Gray color; acc = fn acc u})
            (Char_map.find u g)
        in
        let t = t + 1 in
        t , {d; f=(Char_map.add u t f); pred;
             color=Char_map.add u Black color; acc}
      in
      let v = List.fold_left (fun k (x, _) -> x :: k) []
                             (Char_map.bindings g) in
      let initial_state =
        {d=Char_map.empty;
         f=Char_map.empty;
         pred=Char_map.empty;
         color=List.fold_right (fun x->Char_map.add x White)
                               v Char_map.empty;
         acc=acc}
      in
      (snd (dfs_visit 0 c initial_state)).acc
  end;;
module M : GRAPH

# let g = M.of_adjacency
            ['u', ['v'; 'x'];
             'v',      ['y'];
             'w', ['z'; 'y'];
             'x',      ['v'];
```

```
             'y',      ['x'];
             'z',      ['z'];
            ];;
val g : M.t = <abstr>
# List.rev (M.dfs_fold g 'w' (fun acc c -> c :: acc) []);;
- : M.node list = ['w'; 'z'; 'y'; 'x'; 'v']
```

Connected components. (*medium*)

Write a predicate that splits a graph into its connected components.

```
(* example pending *);;
```

Bipartite graphs. (*medium*)

Write a predicate that finds out whether a given graph is bipartite.

```
(* example pending *);;
```

Generate K-regular simple graphs with N nodes. (*hard*)

In a K-regular graph all nodes have a degree of K; i.e. the number of edges incident in each node is K. How many (non-isomorphic!) 3-regular graphs with 6 nodes are there?

See also the table of results.

```
(* example pending *);;
```

Miscellaneous Problems

Eight queens problem. (*medium*)

This is a classical problem in computer science. The objective is to place eight queens on a chessboard so that no two queens are attacking each other; i.e., no two queens are in the same row, the same column, or on the same diagonal.

Hint: Represent the positions of the queens as a list of numbers 1..N. Example: `[4;2;7;3;6;8;5;1]` means that the queen in the first column is in row 4, the queen in the second column is in row 2, etc. Use the generate-and-test paradigm.

```
(* example pending *);;
```

Knight's tour. (*medium*)

Another famous problem is this one: How can a knight jump on an N×N chessboard in such a way that it visits every square exactly once?

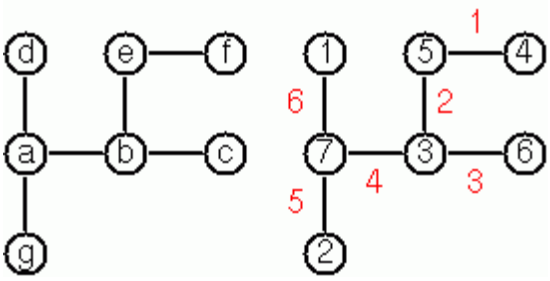
Hints: Represent the squares by pairs of their coordinates `(x,y)`, where both `x` and `y` are integers between 1 and N. Define the function `jump n (x,y)` that returns all coordinates `(u,v)` to which a knight can jump from `(x,y)` to on a `n`×`n` chessboard. And finally, represent the solution of our problem as a list knight positions (the knight's tour).

```
(* example pending *);;
```

Von Koch's conjecture. (*hard*)

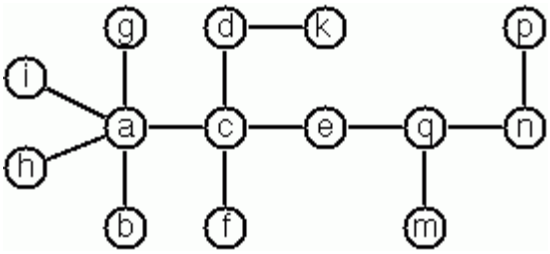
Several years ago I met a mathematician who was intrigued by a problem for which he didn't know a solution. His name was Von Koch, and I don't know whether the problem has been solved since.

Anyway, the puzzle goes like this: Given a tree with N nodes (and hence N-1 edges). Find a way to enumerate the nodes from 1 to N and, accordingly, the edges from 1 to N-1 in such a way, that for each edge K the difference of its node numbers equals to K. The conjecture is that this is always possible.



For small trees the problem is easy to solve by hand. However, for larger trees, and 14 is already very large, it is extremely difficult to find a solution. And remember, we don't know for sure whether there is always a solution!

Write a function that calculates a numbering scheme for a given tree. What is the solution for the larger tree pictured here?




```
(* example pending *);;
```

An arithmetic puzzle. (*hard*)

Given a list of integer numbers, find a correct way of inserting arithmetic signs (operators) such that the result is a correct equation. Example: With the list of numbers `[2;3;5;7;11]` we can form the equations 2-3+5+7 = 11 or 2 = (3*5+7)/11 (and ten others!).

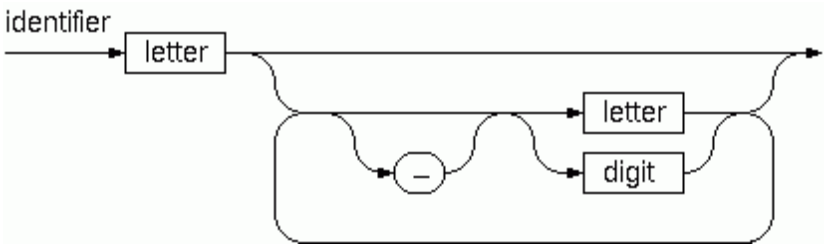
```
(* example pending *);;
```

English number words. (*medium*)

On financial documents, like cheques, numbers must sometimes be written in full words. Example: 175 must be written as one-seven-five. Write a function `full_words` to print (non-negative) integer numbers in full words.

```
(* example pending *);;
```

Syntax checker. (*medium*)



In a certain programming language (Ada) identifiers are defined by the syntax diagram (railroad chart) opposite. Transform the syntax diagram into a system of syntax diagrams which do not contain loops; i.e. which are purely recursive. Using these modified diagrams, write a function `identifier : string -> bool` that can check whether or not a given string is a legal identifier.

```
(* example pending *);;
```

Sudoku. (*medium*)

Sudoku puzzles go like this:

```
Problem statement               Solution

 .  .  4 | 8  .  . | .  1  7     9  3  4 | 8  2  5 | 6  1  7
         |         |                    |          |
 6  7  . | 9  .  . | .  .  .     6  7  2 | 9  1  4 | 8  5  3
         |         |                    |          |
 5  .  8 | .  3  . | .  .  4     5  1  8 | 6  3  7 | 9  2  4
--------+---------+--------     --------+----------+--------
 3  .  . | 7  4  . | 1  .  .     3  2  5 | 7  4  8 | 1  6  9
         |         |                    |          |
 .  6  9 | .  .  . | 7  8  .     4  6  9 | 1  5  3 | 7  8  2
         |         |                    |          |
 .  .  1 | .  6  9 | .  .  5     7  8  1 | 2  6  9 | 4  3  5
--------+---------+--------     --------+----------+--------
 1  .  . | .  8  . | 3  .  6     1  9  7 | 5  8  2 | 3  4  6
         |         |                    |          |
 .  .  . | .  .  6 | .  9  1     8  5  3 | 4  7  6 | 2  9  1
         |         |                    |          |
 2  4  . | .  .  1 | 5  .  .     2  4  6 | 3  9  1 | 5  7  8
```

Every spot in the puzzle belongs to a (horizontal) row and a (vertical) column, as well as to one single 3x3 square (which we call "square" for short). At the beginning, some of the spots carry a single-digit number between 1 and 9. The problem is to fill the missing spots with digits in such a way that every number

between 1 and 9 appears exactly once in each row, in each column, and in each square.

```
(* example pending *);;
```

Nonograms. (*hard*)

Around 1994, a certain kind of puzzles was very popular in England. The "Sunday Telegraph" newspaper wrote: "Nonograms are puzzles from Japan and are currently published each week only in The Sunday Telegraph. Simply use your logic and skill to complete the grid and reveal a picture or diagram." As an OCaml programmer, you are in a better situation: you can have your computer do the work!

The puzzle goes like this: Essentially, each row and column of a rectangular bitmap is annotated with the respective lengths of its distinct strings of occupied cells. The person who solves the puzzle must complete the bitmap given only these lengths.

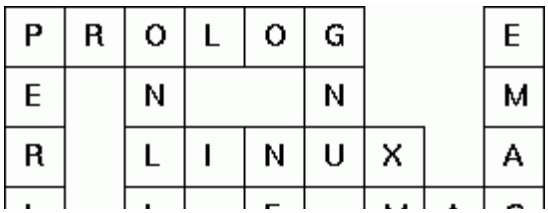
```
Problem statement:          Solution:

|_|_|_|_|_|_|_|_| 3        |_|X|X|X|_|_|_|_| 3
|_|_|_|_|_|_|_|_| 2 1      |X|X|_|X|_|_|_|_| 2 1
|_|_|_|_|_|_|_|_| 3 2      |_|X|X|X|_|_|X|X| 3 2
|_|_|_|_|_|_|_|_| 2 2      |_|_|X|X|_|_|X|X| 2 2
|_|_|_|_|_|_|_|_| 6        |_|_|X|X|X|X|X|X| 6
|_|_|_|_|_|_|_|_| 1 5      |X|_|X|X|X|X|X|_| 1 5
|_|_|_|_|_|_|_|_| 6        |X|X|X|X|X|X|_|_| 6
|_|_|_|_|_|_|_|_| 1        |_|_|_|_|X|_|_|_| 1
|_|_|_|_|_|_|_|_| 2        |_|_|_|X|X|_|_|_| 2
 1 3 1 7 5 3 4 3            1 3 1 7 5 3 4 3
 2 1 5 1                    2 1 5 1
```

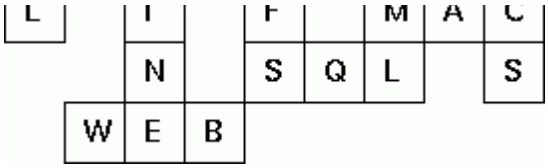
For the example above, the problem can be stated as the two lists `[[3];[2;1];[3;2];[2;2];[6];[1;5];[6];[1];[2]]` and `[[1;2];[3;1];[1;5];[7;1];[5];[3];[4];[3]]` which give the "solid" lengths of the rows and columns, top-to-bottom and left-to-right, respectively. Published puzzles are larger than this example, e.g. 25×20, and apparently always have unique solutions.

```
(* example pending *);;
```

Crossword puzzle. (*hard*)

Given an empty (or almost empty) framework of a crossword puzzle and a set of words. The problem is to place the words into the framework.





The particular crossword puzzle is specified in a text file which first lists the words (one word per line) in an arbitrary order. Then, after an empty line, the crossword framework is defined. In this framework specification, an empty character location is represented by a dot (.). In order to make the solution easier, character locations can also contain predefined character values. The puzzle above is defined in the file p7_09a.dat, other examples are p7_09b.dat and p7_09d.dat. There is also an example of a puzzle (p7_09c.dat) which does not have a solution.

Words are strings (character lists) of at least two characters. A horizontal or vertical sequence of character places in the crossword puzzle framework is called a site. Our problem is to find a compatible way of placing words onto sites.

Hints:

1. The problem is not easy. You will need some time to thoroughly understand it. So, don't give up too early! And remember that the objective is a clean solution, not just a quick-and-dirty hack!
2. For efficiency reasons it is important, at least for larger puzzles, to sort the words and the sites in a particular order.

```
(* example pending *);;
```

Learn

- Code Examples
- Tutorials
- Books
- Success Stories
-

Documentation

- Install
- Manual
- Packages
- Compiler Sources
- Logos

Community

- Mailing Lists
- Meetings
- News
- Support
- Bug Tracker

Contact

- Feedback
- About This Site
- Find Us on GitHub
- Credits