

---

# Table of Contents

Introduction	1.1
Learning Track	1.2
Simple List Functions	1.2.1
Passing functions as arguments	1.2.2
Maybe, Just, Nothing	1.2.3
Recursion	1.2.4
Recursion on Lists	1.2.5
Recursion on Binary Trees	1.2.6
Map	1.2.7
Fold	1.2.8
Problems by Category	1.3
Lists	1.3.1
Lists and Tuples	1.3.2
Binary Trees	1.3.3
Combinations, Randomness, Sorting	1.3.4
Arithmetic	1.3.5
Logic and Codes	1.3.6
Multi-way Trees	1.3.7
Graphs	1.3.8
Puzzles	1.3.9
All Problems	1.4
Problem 1	1.4.1
Problem 2	1.4.2
Problem 3	1.4.3
Problem 4	1.4.4
Problem 5	1.4.5
Problem 6	1.4.6

---

Problem 7	1.4.7
Extra 1	1.4.8
Extra 2	1.4.9
Problem 8	1.4.10
Problem 9	1.4.11
Problem 10	1.4.12
Problem 11	1.4.13
Problem 12	1.4.14
Problem 14	1.4.15
Problem 15	1.4.16
Problem 16	1.4.17
Problem 17	1.4.18
Problem 18	1.4.19
Problem 19	1.4.20
Problem 20	1.4.21
Problem 21	1.4.22
Problem 22	1.4.23
Problem 23	1.4.24
Problem 24	1.4.25
Problem 26	1.4.26
Problem 28a	1.4.27
Problem 28b	1.4.28
Problem 31	1.4.29
Problem 32	1.4.30
Problem 33	1.4.31
Problem 34	1.4.32
Problem 35	1.4.33
Problem 36	1.4.34
Problem 37	1.4.35
Problem 38	1.4.36

---

---

Problem 39	1.4.37
Problem 40	1.4.38
Problem 41	1.4.39
Problem 46	1.4.40
Problem 47	1.4.41
Problem 49	1.4.42
Problem 50a	1.4.43
Problem 50b	1.4.44
Problem 54	1.4.45
Problem 55	1.4.46
Problem 56	1.4.47
Problem 57	1.4.48
Problem 61a	1.4.49
Problem 61b	1.4.50
Problem 62a	1.4.51
Problem 62b	1.4.52
Problem 63	1.4.53
Problem 64	1.4.54
Problem 65	1.4.55
Problem 67	1.4.56
Problem 67b	1.4.57
Problem 68a	1.4.58
Problem 68b	1.4.59
Problem 68c	1.4.60
Problem 69	1.4.61
Problem 70a	1.4.62
Problem 70b	1.4.63
Problem 71	1.4.64
Problem 72	1.4.65
Problem 73	1.4.66

---

---

Problem 80a	1.4.67
Problem 80b	1.4.68
Problem 81	1.4.69
Problem 84	1.4.70
Problem 86a	1.4.71
Problem 86b	1.4.72
Problem 87a	1.4.73
Problem 87b	1.4.74
Problem 88	1.4.75
Problem 90	1.4.76
Problem 91	1.4.77
Problem 92	1.4.78
Problem 93	1.4.79
Problem 94	1.4.80
Problem 95	1.4.81
Problem 96	1.4.82
Problem 97	1.4.83
Problem 98	1.4.84
Problem 99	1.4.85
All Solutions	1.5
Problem 1 Solutions	1.5.1
Problem 2 Solutions	1.5.2
Problem 3 Solutions	1.5.3
Problem 4 Solutions	1.5.4
Problem 5 Solutions	1.5.5
Problem 6 Solutions	1.5.6
Problem 7 Solution	1.5.7
Extra 1 Solutions	1.5.8
Extra 2 Solutions	1.5.9
Problem 8 Solutions	1.5.10

---

---

Problem 9 Solutions	1.5.11
Problem 10 Solutions	1.5.12
Problem 11 Solutions	1.5.13
Problem 12 Solutions	1.5.14
Problem 14 Solutions	1.5.15
Problem 15 Solutions	1.5.16
Problem 16 Solutions	1.5.17
Problem 17 Solutions	1.5.18
Problem 18 Solutions	1.5.19
Problem 19 Solution	1.5.20
Problem 20 Solutions	1.5.21
Problem 21 Solutions	1.5.22
Problem 22 Solutions	1.5.23
Problem 23 Solutions	1.5.24
Problem 24 Solution	1.5.25
Problem 26 Solutions	1.5.26
Problem 27 Solutions	1.5.27
Problem 28a Solutions	1.5.28
Problem 28b Solutions	1.5.29
Problem 31 Solutions	1.5.30
Problem 32 Solutions	1.5.31
Problem 33 Solutions	1.5.32
Problem 34 Solutions	1.5.33
Problem 35 Solutions	1.5.34
Problem 36 Solutions	1.5.35
Problem 37 Solutions	1.5.36
Problem 38 Solutions	1.5.37
Problem 39 Solutions	1.5.38
Problem 40 Solutions	1.5.39
Problem 41 Solutions	1.5.40

---

---

Problem 46 Solutions	1.5.41
Problem 47 Solutions	1.5.42
Problem 49 Solutions	1.5.43
Problem 50a Solutions	1.5.44
Problem 54 Solutions	1.5.45
Problem 55 Solutions	1.5.46
Problem 56 Solutions	1.5.47
Problem 57 Solutions	1.5.48
Problem 61a Solutions	1.5.49
Problem 61b Solutions	1.5.50
Problem 62a Solutions	1.5.51
Problem 62b Solutions	1.5.52
Problem 63 Solutions	1.5.53
Problem 64 Solutions	1.5.54
Problem 65 Solutions	1.5.55
Problem 67 Solution	1.5.56
Problem 68a Solutions	1.5.57
Problem 68b Solutions	1.5.58
Problem 68c Solutions	1.5.59
Problem 69 Solutions	1.5.60
Problem 70a Solutions	1.5.61
Problem 70b Solutions	1.5.62
Problem 71 Solutions	1.5.63
Problem 72 Solutions	1.5.64
Problem 73 Solutions	1.5.65
Problem 74 Solutions	1.5.66
Problem 75 Solutions	1.5.67
Problem 76 Solutions	1.5.68
Problem 77 Solutions	1.5.69
Problem 78 Solutions	1.5.70

---

---

Problem 79 Solutions	1.5.71
Problem 80a Solutions	1.5.72
Problem 80b Solutions	1.5.73
Problem 81 Solutions	1.5.74
Problem 84 Solutions	1.5.75
Problem 86a Solution	1.5.76
Problem 86b Solution	1.5.77
Problem 87a Solution	1.5.78
Problem 87b Solution	1.5.79
Problem 88 Solutions	1.5.80

# Ninety-nine Problems, Solved in Elm

## Who is this book for?

Elm is an easy to use pure functional programming language. These exercises give you a path to learn and practice functional idioms. The techniques demonstrated in this book apply to any functional programming language.

This book makes it easy to jump into programming, no installs required. You can code and test your solutions on Elm's online editor, <http://elm-lang.org/try>. To run with a debugger use <http://debug.elm-lang.org/try>. Every problem has a unit test and a full program to test your solution. Many problems have multiple solutions, demonstrating different approaches to solving a problem.

## Who *isn't* this book for?

Elm is an easy to use web development language. The problems posed in this book do not cover web specific topics. If you are comfortable with functional programming and are looking to learn how to use Elm to develop web apps and web pages, then this book is NOT for you. Try instead Evan Czaplicki's [Introduction to Elm](#). However there are two problems that use the [Elm Architecture](#) to handle randomness ([Problem 23](#)) and time ([Problem 38](#)).

## Where to begin

If you're new to functional programming, start with the [Learning Track](#). This will introduce you to the basic techniques of functional programming in a progression to help you learn through practice. If you just want to jump into the problems, start at [Problems by Category](#).

## History of the 99 Problems



These problems are adaptations for Elm from [Ninety-Nine Haskell Problems](#), which are adaptations of [Ninety-Nine Prolog Problems](#) developed by Werner Hett. The title is more figurative than literal. There weren't 99 problems in the original Prolog collection. A few of the original Prolog problems don't apply to Elm. The numbering of the problems is consistent with the other collections to ease comparisons. Other adaptations include [Lisp](#), [Scala](#), [OCaml](#), [Python](#) and [R](#).

# Learning Track

The problems are numbered to match the original *99 Problems in Prolog*. This is not the best order to tackle the problems if you are new, or new-ish to functional programming. This Learning Track presents an order designed to learn functional programming in a sequence that builds from the most basic idioms to more complex. The problems let you develop experience before moving on to the next concept. Each section presents a new technique and problems that you can solve with that technique.

1. Simple List functions
2. Passing functions as arguments
3. Maybe Just Nothing
4. Recursion
5. Recursion on Lists
6. Recursion on Binary Trees
7. Map
8. Fold

Note the Learning Track does not include all of the 99 problem, and sometimes repeats a problem so you can solve it a different way. For example you will solve Problem 4, `countElements`, three different times, with three different approaches.

## Problem presentation

Each section will have links to problems in order from simple to complex. This will often not be in numerical order. A link is provided to each problem page which has:

1. The problem statement: Describes problem, and ask you to implement a solution.
2. An example: Presents a example of the function with input and the correct result.
3. A unit test: A complete program that executes multiple tests of problem

solution. You can use this program on the Elm online editor, <http://elm-lang.org/try>. The stubbed-out function you need to implement will be at the top, just below the imports.

4. Hints: Suggests one or more ways to implement a solution.
5. Solutions: Links to a page with one or more solutions.

## Learning from the problems

Copy the unit test into the Elm online editor, <http://elm-lang.org/try>, or in the debugger-enabled <http://debug.elm-lang.org/try>. Find the stubbed-out functions with comment `-- your implementation goes here`. Attempt to implement the function. You may want or to write helper functions to complete your solutions.

Click "Compile" to test your code. If your solution is correct you will be rewarded with the message "Your implementation passed all tests". If tests fail, examine which tests failed and try again.

We learn best by doing, so before checking the hints or the solutions page, attempt to solve the problem yourself. Apply what you learned from the previous problems to solve the later problems. Then check the hints and see if that doesn't inspire other solutions or improvements to your own. Finally, check the solutions page to learn from those examples.

If you prefer to install Elm and compile your programs then include the module statement at the top of the test program.

```
module Main exposing (..)
```

## Simple List functions

Let's learn some basic functions from the `List` module.

### Problems to solve using simple List functions

[Problem 6](#) - Use `List.reverse` to test if a list is a palindrome.

[Problem 17](#) - Use `List.take` and `List.drop` to split a list into two lists.

[Problem 18](#) - Use `List.take` and `List.drop` to take a portion of a list

[Problem 20](#) - Use `List.take` and `List.drop` and the append operator `(++)` or `List.append` to remove an element from a list.

[Problem 19](#) - Use `List.take` and `List.drop` and the append operator `(++)` or `List.append` to rotate the elements of a list.

[Problem 21](#) Use `List.take` , `List.drop` , append operator `(++)` and the "cons" operator `:::` to get a specified element from a list. You may want to use split from [Problem 17](#).

## Passing functions as arguments

In many programming languages functions are first-class. This means that a function is a value that can be passed as a argument to other functions. Functions are as fundamental as integers, characters, booleans and strings. The problems below demonstrate functions in the Elm core that take functions are arguments. Likewise, you will often create your own functions that take functions are arguments.

## Functions that take functions

**Problem 28a** - Use `List.sortBy` to sort a list.

**Problem 31** - Use `List.filter` to implement the Sieve of Eratosthenes.

**Problem 34** - Use `List.filter` to implement Euler's totient function.

# Maybe, Just, Nothing

There is no first element or last element of an empty list. So what should a self-respecting programming language return? It might return such a `NULL` or `undefined` and trust the programmer to check for that value. It might raise an exception it requires or maybe just hopes the programmer will handle.

Elm promises to have no runtime exceptions. To guarantee that the application properly handles such conditions Elm borrows the `Maybe` type from Haskell. Let's take a look at its definition.

```
Type Maybe a = Just a | Nothing.
```

We see `Maybe` is a union type. A `Maybe` must either be a `Just` something or a `Nothing`. Also it is parameterized, so we can use it for lists of any type.

## Problems returning Maybe

[Problem 1](#) - Get last element of a list.

[Problem 2](#) - Get penultimate element of a list.

[Problem 3](#) - Get the element of a list at a specified index.

# Recursion

A recursive function is a function that calls itself until it arrives at the solution.

## Problems to solve with recursion

[Problem 32](#) - Use recursion to implement Euclid's algorithm to find the greatest common denominator of two numbers.

[Problem 31](#) - Use recursion to implement the Sieve of Eratosthenes.

# Recursion on Lists

## Example 1

Let's use [Problem 4](#) as a simple example of recursing through a list.

```
countElements : List a -> Int
countElements list =
  case list of
    [ ]
      -> 0

    hd :: tl
      -> 1 + countElements tl
```

When recursing over a list's items you will often use the `x :: xs` cons construction to identify the head and the tail. You can use the head as a value then pass the tail to the same function. In this example we ignore the list item value, adding 1 for each element regardless of its value.

To avoid an infinite loop, there must be a case where the function is not called, bringing an end to the recursion stack. Frequently, as in this example, that is the empty list.

## Example 2

Frequently when recursing over a list, you will build up a new list to return as a result. [Problem 5](#) builds a new list in reverse order from its input.



```
myReverse : List a -> List a
myReverse list =
  case list of
    [] ->
      []

    x :: xs ->
      (myReverse xs) ++ [x]
```

## Problems to solve using recursion

**Problem 14** - Use recursion to duplicate each item of a list using recursion and `(::)` .

**Problem 15** - Use recursion to repeat a specified number of times each item of a list using recursion and `(::)` and `List.repeat` .

**Extra 1** - **Pass a function** to remove some elements from the front of a list.

**Extra 2** - Pass a function to remove some elements from the back of a list.

**Problem 8** - Use `List.take` and `List.drop` to take a portion of a list

# Recursion on Binary Trees

## Binary Trees

### Problems

[Problem 68a](#) - Extract a list of nodes of a tree in pre-order.

[Problem 68b](#) - Extract a list of nodes of a tree in in-order.

[Problem 68c](#) - Extract a list of nodes of a tree in post-order.

[Problem 57](#) - Generate a binary tree from a list of integers.

[Problem 61a](#) - Count the leaves of a tree.

[Problem 61b](#) - Collect the leaves of tree into a list.

[Problem 62a](#) - Count the internal nodes of a tree.

[Problem 62b](#) - Collect the internal nodes of tree into a list.

[Problem 63](#) - Construct a complete binary tree.

[Problem 64](#) - Give coordinates to draw a binary tree

[Problem 70](#) - Generate a dot-string representation of a tree.

# Map

`List.map` applies a provided function to all elements of a list.

## Example

```
List.map negate [1, 2, -3] == [-1, -2, 3]
List.map abs [1, 2, -3] == [1, 2, 3]
```

## Problems to solve using List.map

[Problem 4](#) - Count the elements using map. Hint: after passing a simple function to map, you can get the count by calling `List.sum`.

[Problem 10](#) - Convert a list of lists to a list of lengths.

## ConcatMap

`List.concatMap` applies a function that returns a list to the elements of a list, then concatenates the resulting list of lists into a single list.

[Problem 14](#) - Duplicate each element of a list.

[Problem 15](#) - Repeat elements of a list.

[Problem 7](#) - Flatten nested lists.

[Problem 12](#) - Decode a run-length encoded list.

[Problem 72](#) - Collect the nodes of a multiway tree into a list.

# Fold

Also called reduce in some languages, a fold goes through elements of a list and returns a single value. Folds require a function passed to it, which will be applied to each element of the list. This function can be a named, or an anonymous function. Anonymous functions are also called lambda functions. Key to learning to use folds is thinking of what the function you will pass to fold takes as an argument, and what it should return. You can fold from left to right through a list, `List.foldl`, or from right to left, `List.foldr`. In general you should prefer `foldl` over `foldr` because it is more efficient.

## Problems to solve using List.foldl

**Problem 5** - Use `List.foldl` to reverse a list. Think about the function you want to pass to `foldl`. Its first argument will be the same type as the elements of the list you folding. The second argument will be the same type that we want to return. So, what do we want to return? In this case our goal is to arrive as a List, so let's return a List. Now we know the form of the function we want to pass to `List.foldl`, it should be of this type.

```
foo : a -> List a -> List a
```

Before you can write a function that will build our reversed list, is there a built in function you can use? What functions do we know that have this type?  
(Remember, operators are functions too!)

**Problem 1** - Use `foldl` to find the last element of a list.

**Problem 4** - Count the elements of a list.

**Problem 8** - Remove duplicates from a list.

**Problem 14** - Duplicate list elements.

**Problem 57** - Construct a binary search tree.



# The Problems by Category

This section presents the 99 problems as they were first presented in 99 Problems in Prolog, arranged in topical groups. Each group will have links to the full problem pages.

## Problem pages

Each problem page has:

1. Problem statement: Describes problem, and ask you to implement a solution.
2. Example: Presents a example of the function with input and the correct result.
3. Unit Test: A complete program that executes multiple tests of problem solution. You can use this program on the Elm online editor, <http://elm-lang.org/try>. The stubbed-out function you need to implement will be at the top, just below the imports.
4. Hints: Suggests one or more ways to implement a solution.
5. Solutions: Links to a page with one or more solutions.

## How to get the most from this book

Copy the unit test into the Elm online editor, <http://elm-lang.org/try>. Before checking the hints or the solutions page, attempt to solve the problem by yourself. Apply what you learned from the previous problems to solve the later problems. Then check the hints and see if that doesn't inspire other solutions or improvements to your own. Finally, check the solutions page to learn from those examples.

# Lists

Lists are a fundamental data of functional programming languages. In statically typed languages like Elm all elements of the list are of the same type. Lists are immutable. You never change the list's value. Instead you create new lists, sometimes as modified versions of an input list.

[Problem 1](#) - Get last element of a list.

[Problem 2](#) - Get penultimate element of a list.

[Problem 3](#) - Get the element of a list at a specified index.

[Problem 4](#) - Get the count of elements in a list.

[Problem 5](#) - Reverse a list.

[Problem 6](#) - Determine if a list is a palindrome.

[Problem 7](#) - Flatten nested lists.

---

## List Extras

Many of the solutions to problems in this book can take advantage of some commonly used list manipulation functions. The Elm community created the `List.Extra` module which is well worth your time to peruse. These functions will be familiar to experienced functional programmers, especially Haskell programmers.

Since `List.Extra` is not part of the Elm core we won't use the module in this book. However some of the functions are inescapably useful. So let's add a few extra problems to the original 99, so we can use them in later solutions.

[Extra 1](#) - `dropWhile`

[Extra 2](#) - `takeWhile`

---

## More problems

[Problem 8](#) - Eliminate consecutive duplicate elements from a list.

[Problem 9](#) - Place consecutive duplicate elements of a list into a list of lists.



# Lists and Tuples

## What's a Tuple

A Tuple is a fixed length collection of elements. Unlike Lists, a tuple can mix types. The type of the tuple depends on the types of its elements. Thus a

```
(Int, String)
```

is a different type from a

```
(String, Int)
```

is different from a

```
(Int, String, String, String)
```

## Problems

[Problem 10](#) - Find runs in a list.

[Problem 11](#) - Run length encode a list with tuples.

[Problem 12](#) - Decode run lengths.

[Problem 14](#) - Duplicate elements of a list.

[Problem 15](#) - Replicate elements a given number of times.

[Problem 16](#) - Drop every nth element from a list.

[Problem 17](#) - Split a list into two lists.

[Problem 18](#) - Extract a sublist.

[Problem 19](#) - Rotate the elements in a list.

[Problem 20](#) - Remove the element at a specified index.

[Problem 21](#) - Insert an element at a specified index.

[Problem 22](#) - Create a list of integers in a specified range.



# Binary Trees

---

## Grow a tree

- [Problem 55](#) - Generate balanced binary trees.
  - [Problem 56](#) - Determine if a tree is symmetric.
  - [Problem 57](#) - Generate a binary tree from a list of integers.
  - [Problem 58](#) - Generate symmetric and balanced trees of a given number of nodes.
  - [Problem 59](#) - Generate height-balanced trees.
  - [Problem 60](#) - Generate height-balanced trees with a give number of nodes.
- 

## Traverse a Tree

Solving these three tree traversal problems will make it easier to solve the rest of the tree challenges

- [Problem 68a](#) - Extract a list of nodes of a tree in pre-order.
  - [Problem 68b](#) - Extract a list of nodes of a tree in in-order.
  - [Problem 68c](#) - Generate a tree from a pre-order list of nodes.
- 

## Welcome to the Jungle

- [Problem 61a](#) - Count the leaves of a tree.
  - [Problem 61b](#) - Collect the leaves of tree into a list.
  - [Problem 62a](#) - Count the internal nodes of a tree.
  - [Problem 62b](#) - Collect the internal nodes of tree into a list.
  - [Problem 63](#) - Construct a complete binary tree.
  - [Problem 64](#) - Give coordinates to draw a binary tree, method 1.
  - [Problem 65](#) - Give coordinates to draw a binary tree, method 2.
-

- [Problem 66](#) - Give coordinates to draw a binary tree, method 3.
- [Problem 67](#) - Generate a string representation of a tree.
- [Problem 69](#) - Generate height-balanced trees with a give number of nodes.
- [Problem 70](#) - Generate a dot-string representation of a tree.

# Combinations, Randomness, Sorting

## Problems

[Problem 23](#) - Pick elements from a list at random.

[Problem 24](#) - Pick lottery numbers.

[Problem 25](#) - Randomly pick an permutation of elements from a list.

[Problem 26](#) - Generate all combinations of a specified number of elements of a list.

[Problem 27](#) - Generate all disjoint sets of elements of a list.

[Problem 28a](#) - Sort a list of lists by their lengths.

[Problem 28b](#) - Sort a list of lists by the frequency of their lengths.

# Arithmetic

## Problems

**Problem 31** - Determine if an integer is prime.

**Problem 32** - Find the greatest common denominator of two positive integers.

**Problem 33** - Determine if two integer are coprime.

**Problem 34** - Calculate Euler's totient function.

**Problem 35** - Find the prime factors of a given positive integer.

**Problem 36** - Find the prime factors and their multiplicity of a given positive integer.

**Problem 37** - Euler's totient function revisited.

**Problem 39** - Generate a list of all primes within a given range of integers.

**Problem 40** - Find two primes that sum to a given even integer greater than 3.  
(Goldbach combinations)

**Problem 41** - Find all Goldbach combinations for even integer integers in a given range.

# Logic and Codes

## Problems

[Problem 46](#) - Define functions for logical operations.

[Problem 47](#) - Display truth tables for logical operations.

[Problem 48](#) - Display truth tables for logical functions with multiple boolean inputs.

[Problem 49](#) - Generate Gray codes for a given bit sizes.

[Problem 50](#) - Generate Huffman codes.

# Multiway Trees

## Problems

[Problem 70](#) - Define a multiway tree data type and count the nodes of a multiway tree.

[Problem 71](#) - Calculate the internal path length of a multiway tree.

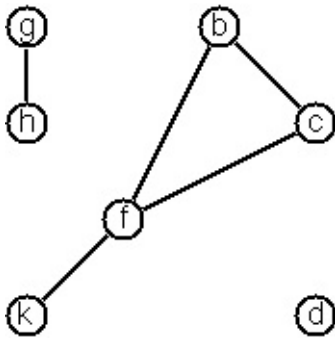
[Problem 72](#) - Generate the bottom-up order list of values from a multiway tree.

[Problem 73](#) - Generate a "lispy" string representation of a multiway tree.



# Graphs

A graph is defined as a set of nodes and a set of edges, where each edge is a pair of nodes. Throughout these problems we will represent nodes with single characters.



There are several ways to represent graphs. One method is to represent each edge separately.

---

## Edge-clause form

```

type alias Edge comparable = (comparable, comparable)

edges80 = [ ('b','c'), ('b','f'), ('c','f'), ('f','k'), ('g','h')
]
```

We call this edge-clause form. The edge-clause cannot represent isolated nodes.

---

## Graph-term form

To represent the whole graph we can use as a pair of sets (nodes and edges).

---

```
type alias Graph comparable = (List comparable, List (Edge comparable))
```

```
graph80 = ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ],
  [ ('b','c'), ('b','f'), ('c','f'), ('f','k'), ('g','h') ] )
```

We call this the graph-term form. It will be our default representation in these problems. Note that the lists should be kept sorted and should not include duplicated elements. Each edge appears only once in the edge list; i.e. an edge `('x', 'y')` represents one edge between x and y. The term `('y', 'x')` is not shown.

### Elm Leaf

*Because these forms require nodes and edges to be sorted and/or have duplicated remove the nodes must be of type `comparable`. This limits us to `number`, `Char`, `String`, lists of comparables or tuples of comparables.*

## Adjacency list form

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:

```
type alias AdjList comparable = List((comparable, List comparable))
adjList80 =
[ ('b', ['c', 'f']),
  ('c', ['b', 'f']),
  ('d', []),
  ('f', ['b', 'c', 'k']),
  ('k', ['f']),
  ('h', ['g'])
]
```

## Human-readable form

Typing the terms by hand is cumbersome and error-prone. We can define a more compact and "human-readable" notation as follows:

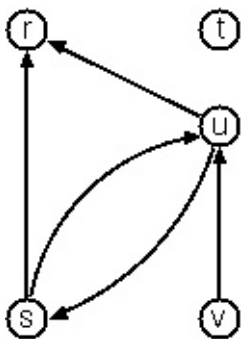
```
hm80 = "[b-c, f-c, g-h, d, f-b, k-f, h-g]"
```

We call this the human-friendly form. The list does not have to be sorted and may even contain the same edge multiple times. Notice the isolated node d. Unlike the Graph and AdjList type, this is textual representation, not a type.

---

## Directed Graphs

When the edges are directed we call them arcs. These are represented by ordered pairs. Such a graph is called directed graph. We can reuse the graph notation with little or no change, to represent digraphs.



```

type alias Arc comparable = (comparable, comparable)

arcs80 = [ ('s','r')
          , ('s','u')
          , ('u','r')
          , ('u','s')
          , ('v','u')]

type Digraph comparable = (List comparable, List (Arc comparable))

type DirAdjList comparable = List((comparable, List comparable))

digraph80 = ( [ 'r', 's', 't', 'u', 'v'], arcs80 )

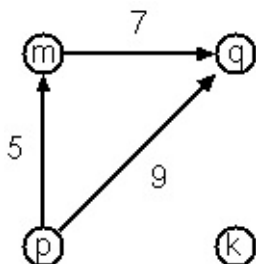
dirAdjList80 = [ (r,[])
                 , (s,[r,u])
                 , (t,[])
                 , (u,[r])
                 , (v,[u])
                 ]

-- human-readable digraph
hrDigraph80 = "[s > r, t, u > r, s > u, u > s, v > u]"

```

## Labeled graphs

Finally, graphs and digraphs may have additional information attached to nodes and edges (arcs). For edges and arc we have to extend our notation. Graphs with additional information attached to edges are called labeled graphs.



```

type alias Arc comparable = (comparable, comparable, Int)
labeledArcs80 =
    [ ('m', 'q', 7)
    , ('p', 'q', 9)
    , ('p', 'm', 5)
    ]

lDigraph80(['k', 'm', 'p', 'q'], labeledArcs80)

type alias LabeledDirAdjList comparable = List((comparable, List
(comparable, Int)))
lDirAdjList80 = [ ('k', [])
                  , ('m', [ ('q', 7) ])
                  , ('p', [ ('m', 5), ('q', 9) ])
                  , ('q', [])
                  ]

lhr80 = "[p>q/9, m>q/7, k, p>m/5]"

```

## Problems

**Problem 80a** - Convert from a graph from graph-term to adjacency-list representation.

**Problem 80b** - Convert from a graph from adjacency-list to graph-term representation.

**Problem 81** - Find the path between two nodes.

**Problem 83** - Generate all spanning trees of a graph.

**Problem 84** - Construct the minimum spanning tree of a graph.

**Problem 85** - Determine if two graphs are isomorphic.

**Problem 86a** - Determine the degree of all nodes of a graph.

**Problem 86b** - Use the Welch-Powell algorithm to color the nodes of a graph such that adjacent nodes have different colors.

**Problem 87a** - Extract the nodes of a graph in depth-first order.

**Problem 87b** - Extract the nodes of a graph in breadth-first order.

**Problem 88** - Split a graph into its connected components.

# Puzzles

## Problems

[Problem 90](#) - Eight Queens.

[Problem 91](#) - Knight's Tour.

[Problem 92](#) - Graceful Labeling.

[Problem 93](#) - Arithmetic puzzle.

[Problem 94](#) - Regular Graphs.

[Problem 95](#) - Number words.

[Problem 96](#) - Syntax checker.

[Problem 97](#) - Sudoku solver.

[Problem 98](#) - Nonogram solver.

[Problem 99](#) - Crossword puzzle solver.

# All Problems

All problems presented in the original order as [Ninety-Nine Prolog Problems](#).



## Problem 1

Write a function `last` that returns the last element of a list. An empty list doesn't have a last element, therefore `last` must return a `Maybe`.

## Example

Cut and paste the following code into the online compiler <http://elm-lang.org/try>. Insert your implementation of `last` and test.

```
import Html exposing (text)
import Maybe

last : List a -> Maybe a
last xs =
  -- your implementation goes here
  Nothing

main =
  case last [ 1, 2, 3, 4 ] of
    Just a ->
      text (toString a)

    Nothing ->
      text "No element found"
```

Result:

```
4
```

## Unit Test

Let's create a more complete test, checking `last` against an empty list and lists of different types. We will use `assertEqual` which takes a tuple, where the first element will hold a test condition and the second holds the expected result. This will allow us to test multiple conditions concisely.

Cut and paste the following code into the online compiler <http://elm-lang.org/try>. Insert your implementation of `last` and test.

```
import Html
import List
import Maybe

last : List a -> Maybe a
last xs =
    -- your implementation goes here
    Nothing

main : Html.Html a
main =
    Html.text
        <| case test of
            0 ->
                "Your implementation passed all tests."

            1 ->
                "Your implementation failed one test."

            x ->
                "Your implementation failed " ++ (toString x) ++
                " tests."

test : Int
test =
    List.length
        <| List.filter ((==) False)
            [ last (1..4) == Just 4
            , last [ 1 ] == Just 1
```

```
    , last [] == Nothing
    , last [ 'a', 'b', 'c' ] == Just 'c'
  ]
```

```
(..) : Int -> Int -> List Int
(..) start end =
  List.range start end
```

## Hints

Here are 2 hints to solve the problem:

1. Use recursion.
2. How can you leverage a List's `head` function to solve this problem?

## Solutions

After you've solved the problem, check out [these solutions](#).

## Problem 2

Implement the function `penultimate` to find the next to last element of a list.

### Example

```
penultimate [1, 2, 3, 4] == Just 3
```

### Unit Test

```
import Html
import List
import Maybe

penultimate : List a -> Maybe a
penultimate list =
  -- your implementation goes here
  Nothing

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."
```

```
test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ penultimate [1, 2, 3, 4] == Just 3
        , penultimate [ 1, 2 ] == Just 1
        , penultimate [ 1 ] == Nothing
        , penultimate [] == Nothing
        , penultimate [ "a", "b", "c" ] == Just "b"
        , penultimate [ "a" ] == Nothing
      ]

(..) : Int -> Int -> List Int
(..) start end =
  List.range start end
```

## Hints

Here are a hints for two different solutions.

1. Use recursion.
2. What can you do to a list to make [List.head](#) solve this problem?

## Solutions

[Solutions](#)

## Problem 3

Implement the function `elementAt` to return the n-th element of a list. The index is 1-relative, that is, the first element is at index 1.

### Example:

```
elementAt [3, 4, 5, 6] 2 == Just 4
```

### Unit Test

```
import Html
import List
import Maybe

elementAt : List a -> Int -> Maybe a
elementAt xs n =
  -- your implementation here
  Nothing

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."
      1 ->
        "Your implementation failed one test."
      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ elementAt [1, 2, 3, 4] 2 == Just 2
      , elementAt [ 1 ] 2 == Nothing
      , elementAt [] 2 == Nothing
      , elementAt [] (-1) == Nothing
      , elementAt [ 'a', 'b', 'c' ] 2 == Just 'b'
      ]
```

## Hints

1. Use recursion.
2. Use `List.drop` .

## Solutions

[Solutions](#)



## Problem 4

Elm provides the function `List.length`. See if you can implement it yourself.

## Example

```
countElements [1, 2, 3, 4, 3, 2, 1] == 7
```

## Unit Test

```
import Html
import List
import Maybe

countElements : List a -> Int
countElements xs =
    -- your implementation here
    0

main : Html.Html a
main =
    Html.text
        <| case test of
            0 ->
                "Your implementation passed all tests."

            1 ->
                "Your implementation failed one test."

            x ->
                "Your implementation failed " ++ (toString x) ++
                " tests."

test : Int
test =
    List.length
        <| List.filter ((==) False)
            [ countElements (List.range 1 4000) == 4000
            , countElements [ 1 ] == 1
            , countElements [] == 0
            , countElements [ 'a', 'b', 'c' ] == 3
            ]
```

## Hints

1. Use a fold.
2. Elm has a specialize fold, `List.sum`. How can you transform the list such that `List.sum` gives you the answer?

## Solutions

### Solutions

## Problem 5

Elm provides the function `List.reverse` to reverse the order of elements in a list. See if you can implement it.

### Example

```
myReverse [1..4] == [4, 3, 2, 1]
```

### Unit Test

```
import Html exposing (text)
import List

myReverse : List a -> List a
myReverse xs =
    -- your implemenation goes here
    []

main : Html.Html a
main =
    Html.text
        <| case test of
            0 ->
                "Your implementation passed all tests."

            1 -> "Your implementation failed one test."

            n -> "Your implementation failed " ++ toString n ++
" tests."

test : Int
test =
    List.length
        <| List.filter ((==) False)
            [ myReverse [1, 2, 3, 4] == [4, 3, 2, 1]
            , myReverse [2, 1] == [1, 2]
            , myReverse [1] == [1]
            , myReverse [] == []
            , myReverse [ 'a', 'b', 'c' ] == [ 'c', 'b', 'a' ]
            ]
```

## Hints

1. Use a fold, passing it a function that takes an element, and returns a list.

## Solutions

## Problem 6

Determine if a list is a palindrome, that is, the list is identical when read forward or backward.

### Example

```
isPalindrome [1,2,3,2,1] == True
```

### Unit Test

```
import Html
import List
import Maybe

isPalindrome : List a -> Bool
isPalindrome xs =
    -- your implementation here
    False

main : Html.Html a
main =
    Html.text
        <| case test of
            0 ->
                "Your implementation passed all tests."

            1 ->
                "Your implementation failed one test."

            x ->
                "Your implementation failed " ++ (toString x) ++
                " tests."

test : Int
test =
    List.length
        <| List.filter ((==) False)
            [ isPalindrome [ 1, 3, 5, 8, 5, 3, 1 ] == True
            , isPalindrome [ 2, 1 ] == False
            , isPalindrome [ 1 ] == True
            , isPalindrome [] == True
            , isPalindrome [ "aa", "bb", "aa" ] == True
            , isPalindrome [ "aab", "b", "aa" ] == False
            ]
```



## Hints

1. Did you solve problem 5?
2. How many elements do you need to test?

## Solutions

[Solutions](#)

## Problem 7

Flatten a nested lists into a single list. Because Lists in Elm are homogeneous we need to define what a nested list is.

```
type NestedList a = Elem a | List [NestedList a]
```

## Example

```
nl1 =  
  SubList  
    [ Elem 1  
    , SubList  
      [ SubList  
        [ Elem 2  
        , SubList [ Elem 3, Elem 4 ]  
        ]  
      , Elem 5  
      ]  
    , Elem 6  
    , SubList [ Elem 7, Elem 8, Elem 9 ]  
    ]  
flatten nl1 == List.range 1 9
```

## Unit Test

```
import Html  
import List  
  
type NestedList a  
  = Elem a  
  | SubList (List (NestedList a))
```

```
flatten : NestedList a -> List a
flatten nl =
    []

main : Html.Html a
main =
    Html.text
        <| case test of
            0 ->
                "Your implementation passed all tests."

            1 ->
                "Your implementation failed one test."

            x ->
                "Your implementation failed " ++ (toString x) ++
" tests."

nl0 =
    SubList [ Elem 1, Elem 2 ]

nl1 =
    SubList
        [ Elem 1
        , SubList
            [ SubList
                [ Elem 2
                , SubList [ Elem 3, Elem 4 ]
                ]
            , Elem 5
            ]
        , Elem 6
        , SubList [ Elem 7, Elem 8, Elem 9 ]
        ]
```

```
test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ flatten n11 == List.range 1 9
        , flatten (SubList [ Elem 1, Elem 2 ]) == [ 1, 2 ]
        , flatten (SubList [ Elem "a", Elem "b" ]) == [ "a",
          "b" ]
        , flatten (SubList [ ]) == [ ]
      ]
```

## Hints

1. You need to use the `case ... of` construct to handle `SubList` and `Elem`.
2. `List.concatMap` will be handy.

## Solutions

[Solutions](#)

## Extra #1 - dropWhile

Drop items from the start of a list until an item does not satisfy criteria specified by a function.

### Example

```
dropWhile isEven [2, 4, 5, 6, 7] == [5, 6, 7]
```

### Unit Test

```
import Html
import List

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
  -- your implementation here
  list

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ ( dropWhile isOdd [1, 2, 1] == [2, 1] )
    , ( dropWhile isEven [1, 2, 1] == [1, 2, 1] )
    , ( dropWhile isEven [] == [] )
    , ( dropWhile isEven [2, 4, 100000, 1] == [1] )
    , ( dropWhile (\x -> x < 5 ) (List.range 1 10) == [5, 6,
7, 8, 9, 10])
    ]

isEven x = x % 2 == 0
isOdd x = x % 2 /= 0
```

## Hints

Recurse!

# Solutions

Solutions

## Extra #2 - takeWhile

Keep elements from the start of a list while they satisfy a condition.

### Example

```
dropWhile isEven [2, 4, 5, 6, 7] == [5, 6, 7]
```

### Unit Test



```
import Html exposing (text)
import List

takeWhile : (a -> Bool) -> List a -> List a
takeWhile predicate list =
    -- your implementation here
    []

main =
    text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all ((==) True)
        [ (takeWhile isOdd [ 1, 2, 1 ] == [ 1 ])
        , (takeWhile isEven [ 1, 2, 1 ] == [])
        , (takeWhile isEven [] == [])
        , (takeWhile isEven [ 2, 4, 100000, 1 ] == [ 2, 4, 100000
])
        , (takeWhile (\x -> x < 5) (List.range 1 10) == [ 1, 2, 3
, 4 ])
        ]

isEven x =
    x % 2 == 0

isOdd x =
    x % 2 /= 0
```

## Hints

1. Recurse!

## Solutions

[Solutions](#)

## Problem 8

Write a function to remove consecutive duplicates of list elements.

### Example

```
noDupses [1, 1, 2, 2, 3, 3, 3, 4, 5, 4, 4, 4, 4]  
        == [1, 2, 3, 4, 5, 4]
```

### Unit Test

```
import Html  
import List  
import Maybe  
  
noDupses : List a -> List a  
noDupses xs =  
    -- your implementation goes here  
    []  
  
main : Html.Html a  
main =  
    Html.text  
        <| case test of  
            0 ->  
                "Your implementation passed all tests."  
  
            1 ->  
                "Your implementation failed one test."  
  
            x ->  
                "Your implementation failed " ++ (toString x) ++  
                " tests."
```

```
test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ noDups [ 1, 1, 1, 1, 2, 5, 5, 2, 1 ] == [ 1, 2, 5
, 2, 1 ]
      , noDups [ 2, 1, 1, 1 ] == [ 2, 1 ]
      , noDups [ 2, 2, 2, 1, 1, 1 ] == [ 2, 1 ]
      , noDups [ 1 ] == [ 1 ]
      , noDups [] == []
      , noDups [ "aa", "aa", "aa" ] == [ "aa" ]
      , noDups [ "aab", "b", "b", "aa" ] == [ "aab", "b",
"aa" ]
      ]
```

## Hints

1. How would you copy a list using `foldr` ? How would you modify the function passed to `foldr` only copy new values?
2. How would you drop the head of the list if it's a duplicate? Can you apply that recursively to solve the problem?

## Solutions

### Solutions

## Problem 9

Convert a list to a list of lists where repeated elements of the source list are packed into sublists. Elements that are not repeated should be placed in a one element sublist.

### Example

```
pack [1,1,1,2,3,3,3,4,4,4,4,5,6,6] ==  
  [ [1,1,1]  
    , [2]  
    , [3, 3, 3]  
    , [4, 4, 4, 4]  
    , [5]  
    , [6, 6]  
  ]
```

### Unit Test

```
import Html  
import List  
import Maybe  
  
pack : List a -> List (List a)  
pack xs =  
  -- your implementation goes here  
  [ [] ]  
  
main : Html.Html a  
main =  
  Html.text  
    <| case test of
```

```

0 ->
    "Your implementation passed all tests."

1 ->
    "Your implementation failed one test."

x ->
    "Your implementation failed " ++ (toString x) ++
    " tests."

test : Int
test =
    List.length
    <| List.filter ((==) False)
        [ pack [ 1, 1, 1, 1, 2, 5, 5, 2, 1 ] == [ [ 1, 1, 1,
1 ], [ 2 ], [ 5, 5 ], [ 2 ], [ 1 ] ]
        , pack [ 2, 1, 1, 1 ] == [ [ 2 ], [ 1, 1, 1 ] ]
        , pack [ 2, 2, 2, 1, 1, 1 ] == [ [ 2, 2, 2 ], [ 1, 1
, 1 ] ]
        , pack [ 1 ] == [ [ 1 ] ]
        , pack [] == []
        , pack [ "aa", "aa", "aa" ] == [ [ "aa", "aa", "aa"
] ]
        , pack [ "aab", "b", "b", "aa" ] == [ [ "aab" ], [ "
b", "b" ], [ "aa" ] ]
        ]

```

## Hints

1. One solution recurses using a helper function accumulating the output lists.
2. `takeWhile` and `dropWhile` could come in handy in another recursive solution.
3. A solution can be written using `foldr`. The function passed to `foldr` needs to return the same output as `pack`, namely `List (List a)`.

## Solutions

## Solutions

## Problem 10

Run-length encode a list of list to a list of tuples. Unlike lists, tuples can mix types. Use tuples `(n, e)` to encode a list where `n` is the number of duplicates of the element `e`.

### Example

```
tuples =  
    [(3, 'a')  
     , (1, 'b')  
     , (3, 'c')  
     , (4, 'd')  
     , (5, 'e')  
     , (6, 'f')]  
  
lists =  
    [ ['a', 'a', 'a']  
      , ['b']  
      , ['c', 'c', 'c']  
      , ['d', 'd', 'd', 'd']  
      , ['e']  
      , ['f', 'f']  
    ]  
  
runLengths lists == tuples
```

### Unit Test

```
import Html  
import List  
import Maybe
```



```

runLengths : List (List a) -> List ( Int, a )
runLengths xss =
    -- your implementation here
    []

main : Html.Html a
main =
    Html.text
        <| case test of
            0 ->
                "Your implementation passed all tests."

            1 ->
                "Your implementation failed one test."

            x ->
                "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
    List.length
        <| List.filter ((==) False)
            [ runLengths [ [ 1, 1, 1, 1 ], [ 2 ], [ 5, 5 ], [ 2
], [ 1 ] ]
                == [ ( 4, 1 ), ( 1, 2 ), ( 2, 5 ), ( 1, 2 ), ( 1
, 1 ) ]
            , runLengths [ [ 2 ], [ 5, 5 ], [ 2 ], [ 1 ] ]
                == [ ( 1, 2 ), ( 2, 5 ), ( 1, 2 ), ( 1, 1 ) ]
            , runLengths [ [ 1, 1, 1, 1 ], [ 2 ], [ 5, 5 ] ]
                == [ ( 4, 1 ), ( 1, 2 ), ( 2, 5 ) ]
            , runLengths [ [ 1, 1, 1, 1 ] ]
                == [ ( 4, 1 ) ]
            , runLengths [ [ "a", "a", "a", "a" ], [ "b" ], [ "c"
, "c" ], [ "b" ], [ "a" ] ]
                == [ ( 4, "a" ), ( 1, "b" ), ( 2, "c" ), ( 1, "b"
), ( 1, "a" ) ]
            , runLengths [ [] ] == []

```

```
    , runLengths [] == []  
  ]
```

## Hints

1. What's a function to convert a list of one type to a list of another type?

## Solutions

[Solutions](#)

## Problem 11

Write a function to run length encode a list, but instead of using a tuple as in problem 10, use a union data type.

```
type RleCode a = Run Int a | Single a
```

## Example

```
rleEncode [1, 1, 1, 1, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5]
  == [Run 4 1, Single 2, Run 2 3, Single 4, Run 6 5]
```

## Unit Test

```
import Html
import List
import Maybe

rleEncode : List a -> List (RleCode a)
rleEncode list =
  -- your implementation here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."
      1 ->
```

```

        "Your implementation failed one test."

    x ->
        "Your implementation failed " ++ (toString x) ++
    " tests."

test : Int
test =
    List.length
    <| List.filter ((==) False)
        [ rleEncode [ 1, 1, 1, 1, 2, 5, 5, 2, 1 ]
            == [ Run 4 1, Single 2, Run 2 5, Single 2, Single
1 ]
        , rleEncode [ 2, 1, 1, 1 ] == [ Single 2, Run 3 1 ]
        , rleEncode [ 2, 2, 2, 1, 1, 1 ] == [ Run 3 2, Run 3
1 ]
        , rleEncode [ 1 ] == [ Single 1 ]
        , rleEncode [] == []
        , rleEncode [ "aa", "aa", "aa" ] == [ Run 3 "aa" ]
        , rleEncode [ "aab", "b", "b", "aa" ]
            == [ Single "aab", Run 2 "b", Single "aa" ]
        ]

```

## Solutions

### Solutions

## Problem 12

Decompress the run-length encoded list generated in [Problem 11](#).

```
rleDecode [Run 4 1, Single 2, Run 2 5, Single 2, Single 1]
== [1, 1, 1, 1, 2, 5, 5, 2, 1]
```

## Unit Test

```
import Html
import List

type RleCode a
  = Run Int a
  | Single a

rleDecode : List (RleCode a) -> List a
rleDecode list =
  -- your implementation goes here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."
      1 ->
        "Your implementation failed one test."
      x ->
        "Your implementation failed " ++ (toString x) ++
```

```
" tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ rleDecode [ Run 4 1, Single 2, Run 2 5, Single 2,
Single 1 ]
        == [ 1, 1, 1, 1, 2, 5, 5, 2, 1 ]
        , rleDecode [ Run 4 1, Single 2, Run 2 5, Single 2,
Single 1 ]
        == [ 1, 1, 1, 1, 2, 5, 5, 2, 1 ]
        , rleDecode [ Run 4 "1", Single "b", Run 2 "5", Sing
le "2", Single "a" ]
        == [ "1", "1", "1", "1", "b", "5", "5", "2", "a"
]
      ]
```

## Hints

1. Check out [List.concatMap](#).

## Solutions

[Solutions](#)

## Problem 14

Duplicate each element of a list.

### Example

```
duplicate [1, 2, 3, 5, 8, 8] == [1, 1, 2, 2, 3, 3, 5, 5, 8, 8, 8, 8]
```

### Unit Test

```
import Html exposing (text)
import List

duplicate : List a -> List a
duplicate list =
    -- your implementation goes here
    []

main =
    text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all (\(result, expect) -> result == expect)
        [ ( duplicate [1, 2, 3, 5, 8, 8], [1, 1, 2, 2, 3, 3, 5,
5, 8, 8, 8, 8] )
        , ( duplicate [], [] )
        , ( duplicate [1], [1, 1] )
        ]
    && List.all (\(result, expect) -> result == expect)
        [ ( duplicate ["1", "2", "5"],
            ["1", "1", "2", "2", "5", "5"] )
        ]
```

## Hints

1. Recurse with cons.

## Solutions



Solutions

## Problem 15

Repeat each element of a list a given number of times.

### Example

```
repeatElements 3 [1, 2, 3, 3] == [1, 1, 1, 2, 2, 2, 3, 3, 3, 3,
3, 3]
```

### Unit Test

```
import Html
import List

repeatElements : Int -> List a -> List a
repeatElements count list =
  -- your implementation goes here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."
```

1. Do I need to *repeat* myself? "Recurse!"
2. Another solution uses List.concatMap
3. Fold!

## Solutions

## Problem 16

Drop every nth element from a list.

### Example

```
dropNth (List.range 1 10) 3 == [1, 2, 4, 5, 7, 8, 10]
```

### Unit Test

```
import Html
import List

dropNth : List a -> Int -> List a
dropNth list n =
  -- your implementation goes here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."
```

```
test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ dropNth [ 1, 2, 5, 5, 2, 1 ] 2 == [ 1, 5, 2 ]
        , dropNth (1..20) 3 == [ 1, 2, 4, 5, 7, 8, 10, 11, 1
3, 14, 16, 17, 19, 20 ]
        , dropNth (1..5) 6 == [ 1, 2, 3, 4, 5 ]
        , dropNth (1..5) 0 == [ 1, 2, 3, 4, 5 ]
        , dropNth (1..5) -1 == [ 1, 2, 3, 4, 5 ]
        , dropNth (1..5) 1 == []
        , dropNth [ "1", "2", "3", "4", "5", "6" ] 2 == [ "1
", "3", "5" ]
      ]

(..) : Int -> Int -> List Int
(..) start end =
  List.range start end
```

## Hints

1. This is why [Problem 20](#) goes in front of Problem 16; add recursion, and more checking of your inputs.

## Solutions

[Solutions](#)

## Problem 17

Split a list into two lists. The length of the first part is specified by the caller.

### Example

```
split (List.range 1 10) 3 == ([1, 2, 3], [4, 5, 6, 7, 8, 9, 10])
```

### Unit Test

```
import Html exposing (text)
import List

split : List a -> Int -> (List a, List a)
split list count =
    -- your implementation here
    ([], [])

main =
    text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all (\(result, expect) -> result == expect)
        [ ( split (List.range 1 5) 0, ([], [1, 2, 3, 4, 5]) )
        , ( split (List.range 1 5) 2, ([1, 2], [3, 4, 5]) )
        , ( split (List.range 1 5) 3, ([1, 2, 3], [4, 5]) )
        , ( split (List.range 1 5) 4, ([1, 2, 3, 4], [5]) )
        , ( split (List.range 1 5) 5, ([1, 2, 3, 4, 5], []) )
        , ( split (List.range 1 5) 6, ([1, 2, 3, 4, 5], []) )
        , ( split (List.range 1 5) (-1), ([], [1, 2, 3, 4, 5]) )
        ]
    && List.all (\(result, expect) -> result == expect)
        [ ( split [ "aab", "b", "c", "aa" ] 2, ([ "aab", "b"
],["c", "aa" ]))
        ]
```

## Hints

1. Take a look in the Elm core package List for a couple functions you can *drop*

in.

## Solutions

Solutions



## Problem 18

Extract a slice from a list.

Given a list, return the elements between (inclusively) two indices. Start counting the elements with 1. Indices outside of the list bounds (i.e. negative number, or beyond the length of the list) should be clipped to the bounds of the list.

### Example

```
slice 3 7 (List.range 1 10) == (List.range 3 7)
slice -7 -3 (List.range 1 10) == []
```

### Unit test

```
import Array
import Html
import List

sublist : Int -> Int -> List a -> List a
sublist start end list =
  -- your implementation goes here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."
      1 ->
        "Your implementation failed one test."
      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."

test : Int
test =
  List.length <| List.filter ((==) False)
    [ True
    , sublist 3 7 (List.range 1 10) == List.range 3 7
    , sublist 2 100 [ 'a', 'b', 'c' ] == [ 'b', 'c' ]
    , sublist -1 2 (List.range 1 100) == [1, 2]
    , sublist -3 -2 [-3, -2, -1, 0, 1, 2, 3] == []
    , sublist 5 3 [ "indices", " are", "inverted" ] == []
    , sublist 0 1 (List.range 1 10) == [1]
    , sublist -7 -3 (List.range 1 10) == []
    ]
```

## Hint

1. Before you *drop* in the simplest solution, *take* a minute to consider the *array* of possibilities.

## Solutions

### Solutions

## Problem 19

Rotate a list `n` places to the left (negative values will rotate to the right). Allow rotations greater than the list length.

### Example

```
[rotate 3 (List.range 1 10), rotate 11 (List.range 1 10) ] ==  
  [ [4, 5, 6, 7, 8, 9, 10, 1, 2, 3]  
    , [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]  
  ]
```

### Unit Test

```
import Html  
import List  
  
rotate : Int -> List a -> List a  
rotate list rot =  
  []  
  
main : Html.Html a  
main =  
  Html.text  
    <| case test of  
      0 ->  
        "Your implementation passed all tests."  
      1 ->  
        "Your implementation failed one test."  
      x ->  
        "Your implementation failed " ++ (toString x) ++
```

```
" tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ rotate 3 [ 1, 2, 5, 5, 2, 1 ] == [ 5, 2, 1, 1, 2, 5
]
      , rotate 13 (List.range 1 10) == [ 4, 5, 6, 7, 8, 9,
10, 1, 2, 3 ]
      , rotate 1 (List.range 1 5) == [ 2, 3, 4, 5, 1 ]
      , rotate 0 (List.range 1 5) == [ 1, 2, 3, 4, 5 ]
      , rotate -1 (List.range 1 5) == [ 5, 1, 2, 3, 4 ]
      , rotate -6 (List.range 1 5) == [ 5, 1, 2, 3, 4 ]
      , rotate 3 (List.range 1 5) == [ 4, 5, 1, 2, 3 ]
      , rotate 1 [ "1", "2", "3", "4" ] == [ "2", "3", "4"
, "1" ]
    ]
```

## Hints

1. Take and ...

## Solutions

[Solutions](#)

## Problem 20

Remove the *n*th element from a list.

### Example

```
dropAt 3 (List.range 1 10) == [1, 2, 4, 5, 6, 7, 8, 9, 10]
```

### Unit Test

```
import Html
import List

dropAt : Int -> List a -> List a
dropAt n list =
  -- your implementation here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."
```

```
test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ dropAt 2 [ 1, 2, 5, 5, 2, 1 ] == [ 1, 5, 5, 2, 1 ]
        , dropAt 3 (List.range 1 14) == [ 1, 2, 4, 5, 6, 7, 8
          , 9, 10, 11, 12, 13, 14 ]
          , dropAt 6 (List.range 1 5) == [ 1, 2, 3, 4, 5 ]
          , dropAt 0 (List.range 1 5) == [ 1, 2, 3, 4, 5 ]
          , dropAt -1 (List.range 1 5) == [ 1, 2, 3, 4, 5 ]
          , dropAt 1 (List.range 1 5) == [ 2, 3, 4, 5 ]
          , dropAt 2 [ "1", "2", "3", "4", "5" ] == [ "1", "3"
            , "4", "5" ]
        ]
```

## Solution

[Solution](#)

## Problem 21

Insert an element at a given position into a list. Treat the first position as index 1.

### Example

```
insertAt 2 'l' ['E', 'm'] == ['E', 'l', 'm']
```

### Unit Test

```
import Html
import List

insertAt : Int -> a -> List a -> List a
insertAt n v xs =
  -- your implementation here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."
```



```
test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ insertAt 2 99 [ 1, 2, 5, 5, 2, 1 ] == [ 1, 99, 2, 5
, 5, 2, 1 ]
      , insertAt 3 99 (List.range 1 14) == [ 1, 2, 99, 3, 4
, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 ]
      , insertAt 6 99 (List.range 1 5) == [ 1, 2, 3, 4, 5,
99 ]
      , insertAt 0 99 (List.range 1 5) == [ 99, 1, 2, 3, 4
, 5 ]
      , insertAt -1 99 (List.range 1 5) == [ 99, 1, 2, 3, 4
, 5 ]
      , insertAt 1 99 (List.range 1 5) == [ 99, 1, 2, 3, 4
, 5 ]
      , insertAt 2 "x" [ "1", "2", "3", "4", "5" ] == [ "1"
, "x", "2", "3", "4", "5" ]
    ]
```

## Hints

1. Try using `split` from [Problem 17](#)
2. The core package `List` has what you need.
3. Recursion can solve this.

## Solutions

### Solutions

## Problem 22

Create a list containing all integers within a given range, inclusively, allow for reverse order

### Example

```
range 8 4 == [8, 7, 6, 5, 4]
```

### Unit Test

```
import Html
import List

range : Int -> Int -> List Int
range start end =
  -- your implementation goes here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ range 1 5 == [1, 2, 3, 4, 5]
      , range 0 5 == [0, 1, 2, 3, 4, 5]
      , range -1 5 == [-1, 0, 1, 2, 3, 4, 5]
      , range 5 -1 == [ 5, 4, 3, 2, 1, 0, -1 ]
      , range 5 5 == [ 5 ]
      , List.length (range 1 999) == 999
      ]
```

## Hints

1. Oh, I don't know, maybe recursion?
2. Range only varies from the range operator (..) by allowing reverse order.

## Solutions

[Solutions](#)

## Problem 23

Extract a given number of randomly selected elements from a list.

### Example

```
randomSelect seed 3 ["Al", "Biff", "Cal", "Dee", "Ed", "Flip"] =  
= ["Cal", "Dee", "Al"]
```

You must use [Elm's Random](#) to implement `randomSelect`. Use `Random.step` to generate a pseudo-random number. `Random.step` takes a [Generator](#) and a [Seed](#). The seed is passed as a parameter to `randomSelect`. You will need to create a generator such as [Random.int](#).

`Random.step` will return both a randomly generated value from the generator, and a new seed. You must use the new seed for subsequent random numbers.

### Unit Test

This unit test uses the Elm Architecture to allow a pseudo-random seed to enter into the pure functional world of Elm. You don't need to understand how this works to implement `randomSelect`. Because it uses the [Elm Architecture](#) it will not run on [elm.org/try](http://elm.org/try). Instead, install Elm, compile and run this application on your local machine. See <https://guide.elm-lang.org/install.html>.

```
module Main exposing (..)  
  
import Html exposing (..)  
import Html.Events exposing (..)  
import Random  
  
randomSelect : Random.Seed -> Int -> List a -> ( List a, Random.  
Seed )
```

```
randomSelect seed n list =
  -- your implementation goes here
  ( [], seed )

main : Program Never Model Msg
main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }

-- MODEL

type alias Model =
  { intSeed : Int
  , tested : Bool
  , passed : Bool
  }

init : ( Model, Cmd Msg )
init =
  ( Model 1 False False, Cmd.none )

-- UPDATE

type Msg
  = Test
  | NewFace Int

update : Msg -> Model -> ( Model, Cmd Msg )
```

```
update msg model =
  case msg of
    Test ->
      ( model, Random.generate NewFace (Random.int Random.
minInt Random.maxInt) )

      NewFace newSeed ->
        ( Model newSeed True (test model.intSeed), Cmd.none
)

-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.none

-- VIEW

view : Model -> Html Msg
view model =
  div []
    [ h2 [] [ text (testMsg model.tested model.passed) ]
      , p [] [ text ("Seed value: " ++ (toString (model.intSeed))) ]
      , p [] [ text ("Your die roll is " ++ (toString (Tuple.f
first (randomSelect (Random.initialSeed model.intSeed) 1 (1..6)))
)) ]
      , button [ onClick Test ] [ text "Test" ]
    ]

test : Int -> Bool
test intSeed =
  let
```

```
seed =
  Random.initialSeed intSeed

( l1, s1 ) =
  randomSelect seed 3 (1..1000)

( l2, s2 ) =
  randomSelect seed 3 (1..1000)

( l3, s3 ) =
  randomSelect s2 3 (1..1000)

( l4, s4 ) =
  randomSelect s3 9 (1..9)

( l5, s5 ) =
  randomSelect s4 3 [ "a", "b" ]

( l6, s6 ) =
  randomSelect s5 0 [ 'a', 'b' ]

( l7, s7 ) =
  randomSelect s6 -1 [ 'a', 'b' ]

( l8, s8 ) =
  randomSelect s7 1 [ 'a', 'b' ]
in
List.all ((==) True)
  [ List.sort l1 == List.sort l2
  , l1 == l2
  , l2 /= l3
  , -- a billion to one that this won't match
  , List.sort l4 == 1..9
  , List.sort l5 == [ "a", "b" ]
  , l6 == []
  , l7 == []
  ]

testMsg : Bool -> Bool -> String
```



```
testMsg tested passed =  
  if tested then  
    if passed then  
      "Your implementation passed all tests."  
    else  
      "Your implementation failed at least one test."  
  else  
    "Click the test button below"  
  
(..) : Int -> Int -> List Int  
(..) start end =  
  List.range start end
```

## Hints

1. `elementAt` and `dropAt` from [Problem 3](#) and [Problem 20](#) could prove useful.

## Solutions

[Solutions](#)

## Problem 24

Draw n different random numbers from a range of numbers.

```
lotto seed 6 6 49
```

Example result:

```
[14, 19, 28, 31, 33, 48]
```

## Unit Test

```
import Html exposing (..)
import Html.App as App
import Html.Events exposing (..)
import Random

lotto : Random.Seed -> Int -> Int -> Int -> List Int
lotto seed n low high =
    -- your implementation goes here
    []

main =
    App.program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }

-- MODEL
```

```
type alias Model =
  { intSeed : Int
  , tested : Bool
  , passed : Bool
  }

init : ( Model, Cmd Msg )
init =
  ( Model 1 False False, Cmd.none )

-- UPDATE

type Msg
  = Test
  | NewFace Int

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    Test ->
      ( model, Random.generate NewFace (Random.int Random.
minInt Random.maxInt) )

      NewFace newSeed ->
        ( Model newSeed True True, Cmd.none )

-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
```

Sub.none

-- VIEW

```
view : Model -> Html Msg
view model =
  div []
    [ p [] [ text ("Seed value: " ++ (toString (model.intSeed))) ]
      , p [] [ text ("Your lotto numbers are " ++ (toString (lotto (Random.initialSeed model.intSeed) 6 6 49)))) ]
      , button [ onClick Test ] [ text "Test" ]
    ]

(..) : Int -> Int -> List Int
(..) start end =
  List.range start end
```

## Hints

1. A very simple solution is possible using the solution from [Problem 23](#).

## Solutions

[Solutions](#)

## Problem 26

In how many ways you choose a committee of 3 from a group of 12 people? The [combination formula](#) tells us  $C(12,3) = 220$  possibilities.

Write a function to generate all combinations of a list.

```
combinations 2 ['a', 'b', 'c'] == [['a', 'b'], ['a', 'c'], ['b', 'c']]
```

## Unit Test

```
import Html
import List

combinations : Int -> List a -> List (List a)
combinations n list =
  -- your implementation goes here
  [ [] ]

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."
      1 ->
        "Your implementation failed one test."
      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."
```

```

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ combinations 1 (List.range 1 5) == [ [ 1 ], [ 2 ],
        [ 3 ], [ 4 ], [ 5 ] ]
        , combinations 2 [ 'a', 'b', 'c' ] == [ [ 'a', 'b' ]
        , [ 'a', 'c' ], [ 'b', 'c' ] ]
        , combinations 2 (List.range 1 3) == [ [ 1, 2 ], [ 1
        , 3 ], [ 2, 3 ] ]
        , combinations 2 (List.range 1 4) == [ [ 1, 2 ], [ 1
        , 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
        , combinations 0 (List.range 1 10) == [ [] ]
        , combinations -1 (List.range 1 10) == [ [] ]
        , List.length (combinations 3 (List.range 1 12)) ==
220
        , List.length (combinations 4 (List.range 1 15)) ==
1365
      ]

```

## Hint

1. Combinations of n number of elements can be generated from combinations of n-1.

## Solution

[Solutions](#)

## Problem 28.a

`List.sort` will sort a list from lowest to highest.

```
List.sort [3, 5, 1, 10 -2] == [-2, 1, 3, 5, 10]
```

When you need other sort logic pass a function to `List.sortBy`.

Sort a list of list by the length of the lists. The order of sublists of the same size is undefined.

## Example

```
lists = [[1],[2],[3,4,5],[6,7,8],[2,3],[4,5],[6,7,8,9,0]]
map List.length (sortByListLengths lists) == [1, 1, 2, 2, 3, 3, 5]
]
```

## Unit Test

```
import Html
import List

sortByListLengths : List (List a) -> List (List a)
sortByListLengths xs =
  -- your implementation goes here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
```

```

0 ->
    "Your implementation passed all tests."

1 ->
    "Your implementation failed one test."

x ->
    "Your implementation failed " ++ (toString x) ++
    " tests."

test : Int
test =
    List.length
        <| List.filter ((==) False)
            [ List.map List.length (sortByListLengths [ [], [ 1
], [1..2], [1..3], [1..4], [1..5] ])
                == [ 0, 1, 2, 3, 4, 5 ]
            , List.map List.length (sortByListLengths [ [] ])
                == [ 0 ]
            , List.map List.length (sortByListLengths [ [], [ 1
], [1..100000], [1..4], [1..3], [1..2] ])
                == [ 0, 1, 2, 3, 4, 100000 ]
            , List.map List.length (sortByListLengths [ [ 14 ],
[ 15 ], [], [ 1 ], [ 12 ], [ 13 ] ])
                == [ 0, 1, 1, 1, 1, 1 ]
            , List.map List.length (sortByListLengths [ [ "a", "
b", "c" ], [ "a", "b" ], [ "a" ] ])
                == [ 1, 2, 3 ]
        ]

(..) : Int -> Int -> List Int
(..) start end =
    List.range start end

```

## Hints



1. If you use `List.sortBy` , you need to pass a function to it. Hmm, a function that returns a comparable value based on a list length? What could it possibly be called?

## Solution

Solution

## Problem 28.b

Sort a list according to the frequency of the sublist length. Place lists with rare lengths first, those with more frequent lengths come later. If the frequency of two or more sublists are equal the any order is acceptable.

### Example

```
(List.map List.length
  <| sortByLengthFrequency
  [[1],[2],[3],[6,7,8],[2,34,5],[]])
== [0, 3, 3, 1, 1, 1]
```

### Unit Test

```
import Html
import List

sortByLengthFrequency : List (List a) -> List (List a)
sortByLengthFrequency xs =
  -- your implementation goes here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."
      1 ->
        "Your implementation failed one test."
```

```

x ->
    "Your implementation failed " ++ (toString x) ++
    " tests."

test : Int
test =
    List.length
        <| List.filter ((==) False)
            [ (List.map List.length
                <| sortByLengthFrequency [ [ 1 ], [ 2 ], [ 3 ],
[ 6, 7, 8 ], [ 2, 34, 5 ], [ ] ]
                )
                == [ 0, 3, 3, 1, 1, 1 ]
            , (List.map List.length
                <| sortByLengthFrequency [ [ 1 ], [ 2 ], [ 3 ],
[ 6 ], [ 2 ], (List.range 1 10) ]
                )
                == [ 100000, 1, 1, 1, 1, 1 ]
            , (List.map List.length
                <| sortByLengthFrequency [ [ 1, 2, 3 ], [ 6, 7, 8
], [ 0 ], [ 2, 3, 5 ] ]
                )
                == [ 1, 3, 3, 3 ]
            , (List.map List.length
                <| sortByLengthFrequency [ [ ] ]
                )
                == [ 0 ]
            ]

(..) : Int -> Int -> List Int
(..) start end =
    List.range start end

```

## Hints

1. First find the frequency for each list length, then use that to sort. You may need to partially apply the length/frequency data to the comparison function you pass to `List.sortBy`.
2. The module `List.extra` ([link](#)), not available on <http://elm-lang.org/try>, provides a `groupBy` function. If you group by length, you could then sort the groups and flatten again.

## Solutions

### Solutions

## Problem 31

Determine whether a given integer number is prime.

Example:

```
isPrime 113 == True
```

## Unit Test

```
import Html

isPrime : Int -> Bool
isPrime n =
    -- your implementation goes here
    False

main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all (\( result, expect ) -> result == expect)
        [ ( isPrime 36, False )
        , ( isPrime 10, False )
        , ( isPrime -1, False )
        , ( isPrime 1, False )
        , ( isPrime 0, False )
        , ( isPrime 120, False )
        , ( isPrime 2, True )
        , ( isPrime 23, True )
        , ( isPrime 6000, False )
        , ( isPrime 7919, True )
        , ( isPrime 7911, False )
        , ( isPrime 63247, True )
        , ( isPrime 63249, False )
        ]
```

## Hints

1. Try the [Sieve of Eratosthenes](#). `List.filter` will come in handy.

# Solutions

Solutions

## Problem 32

Determine the greatest common divisor of two positive integer numbers. Use [Euclid's algorithm](#) which recurses over the following steps:

1. Given two numbers, a and b divide a by b
2. If the remainder of the division is 0, the numerator is the gcd.
3. else divide the demoninator by the remainder and return to step 2.

## Example

```
gcd 36 63 = 9
```

## Unit Test



```
import Html
import List
import Maybe

gcd : Int -> Int -> Int
gcd a b =
    -- Your implementation goes here
    0

main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all (\( result, expect ) -> result == expect)
        [ ( gcd 36 63, 9 )
        , ( gcd 10 25, 5 )
        , ( gcd 120 120, 120 )
        , ( gcd 2 12, 2 )
        , ( gcd 23 37, 1 )
        , ( gcd 45 330, 15 )
        , ( gcd 24528 65934, 6 )
        , ( gcd 120 -120, 120 )
        , ( gcd -2 12, 2 )
        , ( gcd 37 23, 1 )
        ]

(..) : Int -> Int -> List Int
(..) start end =
    List.range start end
```

## Hints

1. This is easy!

## Solutions

## Problem 33

Determine whether two positive integer numbers are coprime. Two numbers are coprime if their greatest common divisor equals 1.

```
coprime 35 64 = True
```

## Unit Test

```
import Html
import List

coprime : Int -> Int -> Bool
coprime a b =
  -- your implementation here
  False

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all (\(result, expect) -> result == expect)
    [ ( coprime 36 63, False )
    , ( coprime 10 25, False )
    , ( coprime 120 120, False )
    , ( coprime 2 12, False )
    , ( coprime 1313 1600, True )
    , ( coprime 23 37, True )
    , ( coprime 45 330, False )
    , ( coprime 24528 65934, False )
    , ( coprime 1600 1313, True )
    , ( coprime -23 37, True )
    , ( coprime 330 45, False )
    , ( coprime -23 37, True )
    , ( coprime -330 -45, False )
    , ( coprime -24528 65934, False )
    ]
```

## Hints

1. Use the function `gcd` from [Problem 32](#).

## Solutions

[Solutions](#)

## Problem 34

Calculate Euler's totient function  $\phi(m)$ .

Euler's totient function  $\phi(m)$  is defined as the number of positive integers  $r$  ( $1 \leq r < m$ ) that are coprime with  $m$ .

### Example

```
let m = 10
then coprimes of m are 1,3,7,9
and totient of m is 4
```

Note the special case: `totient 1 == 1`

### Unit Test

```
import Html
import List

totient : Int -> Int
totient n =
  -- your implementation here
  0

main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."
      1 ->
        "Your implementation failed one test."
```

```
n ->
    "Your implementation failed " ++ (toString n) ++
    " tests."

test : Int
test =
    List.length
    <| List.filter ((==) False)
        [ totient 10 == 4
        , totient 25 == 20
        , totient 120 == 32
        , totient 0 == 0
        , totient 1600 == 640
        , totient 37 == 36
        , totient 330 == 80
        , totient 65934 == 19440
        , totient 1313 == 1200
        , totient 45 == 24
        , totient -23 == 0
        ]
```

## Hints

1. Use `coprime` from [Problem 33](#).

## Solutions

[Solutions](#)

## Problem 35

Determine the prime factors of a given positive integer. Construct a flat list containing the prime factors in ascending order.

Example:

```
primeFactors 315 == [3, 3, 5, 7]
```

## Unit Test



```
import Html
import List
import Maybe

primeFactors : Int -> List Int
primeFactors n =
  -- your implementation here
  []

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all (\( result, expect ) -> result == expect)
    [ ( primeFactors 36, [ 2, 2, 3, 3 ] )
    , ( primeFactors 10, [ 2, 5 ] )
    , ( primeFactors -1, [] )
    , ( primeFactors 1, [] )
    , ( primeFactors 0, [] )
    , ( primeFactors 120, [ 2, 2, 2, 3, 5 ] )
    , ( primeFactors 2, [ 2 ] )
    , ( primeFactors 23, [ 23 ] )
    , ( primeFactors 69146, [ 2, 7, 11, 449 ] )
    , ( primeFactors 9007, [ 9007 ] )
    , ( primeFactors 36028, [ 2, 2, 9007 ] )
    , ( primeFactors 26028, [ 2, 2, 3, 3, 3, 241 ] )
    ]
```

## Hints

1. Use `primesRange` from [Problem 39](#) to give you all the prime numbers from 2 to  $n$ , and search for factors.
2. Try to implement the solution without `primesRange`. Instead use `dropWhile` from [Problem 8](#) to filter to the factors.

## Solutions

### Solutions

## Problem 36

Construct a list containing the prime factors and their multiplicity for a given integer.

### Example

```
primeFactorsM 315 = [(3, 2), (5,1), (7,1)]
```

### Unit Test

```

import Html
import List

primeFactorsM : Int -> List ( Int, Int )
primeFactorsM n =
    -- your implementation here
    []

main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all (\( result, expect ) -> result == expect)
        [ ( primeFactorsM 36, [ ( 2, 2 ), ( 3, 2 ) ] )
        , ( primeFactorsM 10, [ ( 2, 1 ), ( 5, 1 ) ] )
        , ( primeFactorsM -1, [] )
        , ( primeFactorsM 1, [] )
        , ( primeFactorsM 0, [] )
        , ( primeFactorsM 120, [ ( 2, 3 ), ( 3, 1 ), ( 5, 1 ) ]
        )
        , ( primeFactorsM 2, [ ( 2, 1 ) ] )
        , ( primeFactorsM 23, [ ( 23, 1 ) ] )
        , ( primeFactorsM 69146, [ ( 2, 1 ), ( 7, 1 ), ( 11, 1 )
        , ( 449, 1 ) ] )
        , ( primeFactorsM 9007, [ ( 9007, 1 ) ] )
        , ( primeFactorsM 36028, [ ( 2, 2 ), ( 9007, 1 ) ] )
        , ( primeFactorsM 26028, [ ( 2, 2 ), ( 3, 3 ), ( 241, 1
        ) ] )
        ]

```

## Hints

1. Combine code from problems [9](#), [10](#) and [35](#)

## Solutions

[Solutions](#)

## Problem 37

Calculate Euler's totient function  $\phi(m)$  (improved).

See [problem 34](#) 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 [problem 36](#) 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  $\phi(m)$  can be calculated with the following formula:

$$\phi(m) = ((p1 - 1) * p1^{(m1 - 1)}) * ((p2 - 1) * p2^{(m2 - 1)}) * ((p3 - 1) * p3^{(m3 - 1)}) * \dots$$

## Unit Test

```
import Html
import List

phi : Int -> Int
phi n =
  -- your implementation here
  0

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test.")
  )
```

```
test : Bool
test =
  List.all (\( result, expect ) -> result == expect)
    [ ( phi 36, totient 36 )
      , ( phi 10, totient 10 )
      , ( phi 1, totient 1 )
      , ( phi 0, totient 0 )
      , ( phi 120, totient 120 )
      , ( phi 2, totient 2 )
      , ( phi 23, totient 23 )
      , ( phi 69145, totient 69145 )
      , ( phi 9007, totient 9007 )
      , ( phi 36028, totient 36028 )
      , ( phi 26028, totient 26028 )
    ]
```

```
totient : Int -> Int
totient n =
  List.length <| coprimes n
```

```
coprimes : Int -> List Int
coprimes n =
  List.filter (\x -> coprime n x) (List.range 1 n)
```

```
coprime : Int -> Int -> Bool
coprime a b =
  gcd a b == 1
```

```
gcd : Int -> Int -> Int
gcd a b =
  if b == 0 then
    abs a
  else
    gcd b (a % b)
```

## Hints

1. Can you use the special fold `List.product` ?

## 2. Solutions

[Solutions](#)



## Problem 38

The application below measures the time to calculate phi of 10090 using the algorithms from Problem 34 and Problem 37. Because it uses the [Elm Architecture](#) it will not run on [elm.org/try](http://elm.org/try). Instead, install Elm, compile and run this application on your local machine. See <https://guide.elm-lang.org/install.html>.

.

```
import Html exposing (div, button, p, program, text)
import Html.Events exposing (onClick)
import Task
import Time exposing (Time)

{- =====
   Test 1
-}

test1 : Int -> Int
test1 n =
    test totient n

totient : Int -> Int
totient n =
    List.length <| List.filter (\x -> coprime n x) (1..n)

coprime : Int -> Int -> Bool
coprime a b =
    gcd a b == 1

gcd : Int -> Int -> Int
gcd a b =
    if b == 0 then
```

```

        abs a
    else
        gcd b (a % b)

{- =====
Test 2
-}

test2 : Int -> Int
test2 n =
    test phi n

phi : Int -> Int
phi n =
    if n < 1 then
        0
    else
        List.product
            <| List.map (\( p, m ) -> (p - 1) * p ^ (m - 1))
            <| primeFactorsM n

primeFactorsM : Int -> List ( Int, Int )
primeFactorsM n =
    toTuples <| primeFactors n

primeFactors : Int -> List Int
primeFactors n =
    if n < 2 then
        []
    else
        let
            prime =
                Maybe.withDefault 0
                <| List.head

```

```
        <| dropWhile (\x -> n % x /= 0) (2..n)

    in
        prime :: (primeFactors <| n // prime)

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
    case list of
        [] ->
            []

        x :: xs ->
            if (predicate x) then
                dropWhile predicate xs
            else
                list

takeWhile : (a -> Bool) -> List a -> List a
takeWhile predicate xs =
    case xs of
        [] ->
            []

        hd :: tl ->
            if (predicate hd) then
                hd :: takeWhile predicate tl
            else
                []

toTuples : List a -> List ( a, Int )
toTuples fs =
    case fs of
        [] ->
            []

        x :: xs ->
            ( x, List.length (takeWhile (\y -> y == x) fs) )
            :: toTuples (dropWhile (\y -> y == x) fs)
```

```
{- =====
   Timing test app
-}

test : (Int -> Int) -> Int -> Int
test f n =
    List.length
        <| List.map f (1..n)

main =
    Html.program
        { init = init
        , view = view
        , update = update
        , subscriptions = (\_ -> Sub.none)
        }

view model =
    div []
        [ button [ onClick (ExecuteTest 1) ] [ text "Test 1" ]
        , div []
            [ text
                <| case Tuple.first model.testTimes1 of
                    Err msg ->
                        msg

                    Ok s ->
                        case Tuple.second model.testTimes1 of
                            Err msg ->
                                msg

                            Ok e ->
                                ("totient completed in " ++ toSt
ring (e - s) ++ " milliseconds.")
```

```

        , p [] []
        , text
        <| case Tuple.first model.testTimes2 of
            Err msg ->
                msg

            Ok s ->
                case Tuple.second model.testTimes2 of
                    Err msg ->
                        msg

                    Ok e ->
                        ("phi completed in " ++ toString
(e - s) ++ " milliseconds.")
                ]
        , button [ onClick (ExecuteTest 2) ] [ text "Test 2" ]
    ]

type Msg
    = NoOp
    | ExecuteTest Int
    | GetTimeFailure String
    | StartTime Int Time
    | RunTest Int Time
    | EndTest Int Time

type alias TestTimes =
    ( TestTime, TestTime )

type alias TestTime =
    Result String Time

type alias Model =
    { testReps :
        Int
        -- don't hard code this or Elm will memoize the test fun

```

```

ctions
    , testTimes1 : TestTimes
    , testTimes2 : TestTimes
}

init : ( Model, Cmd Msg )
init =
    ( { testReps = 1000
      , testTimes1 = ( Err "Please Run Test 1", Err "No end time yet!" )
      , testTimes2 = ( Err "Please Run Test 2", Err "No end time yet!" )
      }
    , Cmd.none
    )

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        EndTest n time ->
            ( case n of
                1 ->
                    { model | testTimes1 = ( Tuple.first model.testTimes1, Ok time ) }

                2 ->
                    { model | testTimes2 = ( Tuple.first model.testTimes2, Ok time ) }

                _ ->
                    model
            , Cmd.none
            )

        ExecuteTest n ->
            ( model
            , Task.perform (StartTime n) Time.now
            )

```


```
GetTimeFailure msg ->
  ( { model | testTimes1 = ( Err "No start time yet!",
Err "No end time yet!" ) }
  , Cmd.none
  )

NoOp ->
  ( model, Cmd.none )

RunTest n time ->
  ( case n of
    1 ->
      { model
        | testTimes1 = ( Ok time, Err "No end ti
me yet!" )
        , testReps = always model.testReps (test
1 model.testReps)
      }
    2 ->
      { model
        | testTimes2 = ( Ok time, Err "No end ti
me yet!" )
        , testReps = always model.testReps (test
2 model.testReps)
      }
    _ ->
      model
      , Task.perform (EndTest n) Time.now
      )

StartTime n time ->
  ( model
  , Task.perform (RunTest n) Time.now
  )
```

```
(..) : Int -> Int -> List Int  
(..) start end =  
    List.range start end
```





## Problem 39

Construct a list of all prime numbers within a range of integers.

Example:

```
primesInRange 10 20 == [11,13,17,19]
```

## Unit Test

```
import Html
import List

primesInRange : Int -> Int -> List Int
primesInRange low high =
  -- your implementation here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
        " tests."

test : Int
```

```
test =
  List.length
    <| List.filter ((==) False)
      [ primesInRange 1 36 == [2,3,5,7,11,13,17,19,23,29,3
1]
        , primesInRange 1 10 == [2,3,5,7]
        , primesInRange -1 1 == []
        , primesInRange 1 1 == []
        , primesInRange 100 1 == []
        , primesInRange 0 1 == []
        , primesInRange 60 100 == [61,67,71,73,79,83,89,97]
        , primesInRange 4 10 == [5, 7]
        , primesInRange 24 100 == [29,31,37,41,43,47,53,59,6
1,67,71,73,79,83,89,97]
      ]
```

## Hints

1. Modify the solution from [Problem 31](#).

## Solutions

[Solutions](#)

## Problem 40

[Goldbach's conjecture](#) says that every positive even integer greater than 2 is the sum of two prime numbers. Example:  $28 = 5 + 23$ . It has been numerically confirmed up to very large numbers but remains unproven.

Return two prime numbers that sum up to a given even integer.

## Example

```
goldbach 100 == (29, 71)
```

## Unit Test

```
import Html
import List
import Maybe

goldbach : Int -> Maybe (Int, Int)
goldbach n =
  -- your implementation here
  Nothing

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
```

```
test =
  List.all (\n -> testGoldbach n <| goldbach n)
    [ 4, 10, 12, 14, 16, 18, 20, 100, 222, 120, 2444, 24444,
      33336, 71000 ]
    && List.all (\n -> (goldbach n) == Nothing) [-99999, -1,
0, 1, 99, 9999]

testGoldbach : Int -> Maybe (Int, Int) -> Bool
testGoldbach n result =
  case result of
    Nothing -> False

    Just (p1, p2) ->
      if n < 3 && result /= Nothing then
        False
      else if isOdd n then
        False
      else if p1 + p2 /= n then
        False
      else if not (isPrime p1) || not (isPrime p2) then
        False
      else
        True

-- The core library function Arithmetic.isOdd is not available
-- on elm-lang.org/try, so we'll recreate it here
isOdd : Int -> Bool
isOdd n =
  n % 2 /= 0

isPrime : Int -> Bool
isPrime n =
  if n < 2 then
    False
  else
    List.member n <| primesInRange 2 n
```

# Solutions

Solutions

## Problem 41

Find all even numbers within a range which are the sum of two prime numbers that are both greater than a specified threshold.

### Example

```
goldbachGT 1 2000 50 == [(73, 919), (61, 1321), (67, 1789), (61, 1867)]
```

### Unit Test

```
import Html
import List

goldbachGT : Int -> Int -> Int -> List (Int, Int)
goldbachGT low high limit =
  -- your implemenation goes here
  []

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ goldbachGT 1 2000 50 == [(73,919),(61,1321),(67,1789),(
61,1867)]
    , goldbachGT (73 + 919) (73 + 919) 50 == [(73,919)]
    , goldbachGT 1 1000 80 == []
    , goldbachGT 1 200 12 == [(19,79),(13,109),(13,113),(19,
109)]
    ]
```

## Hints

1. Use the solution from [Problem 40](#).

## Solutions

[Solutions](#)





## Problem 46

Define functions to provide the logical binary functions and, or, nand, nor, xor, implies, and equivalent.

```
-- True if and only if a and b are true
and' : Bool -> Bool -> Bool

-- True if either a or b are true
or' : Bool -> Bool -> Bool

-- True if either a or b are false
nand' : Bool -> Bool -> Bool

-- True if and only if a and b are false
nor' : Bool -> Bool -> Bool

-- True if a or b is true, but not if both are true
xor' : Bool -> Bool -> Bool

-- True if a is false or b is true
implies : Bool -> Bool -> Bool

-- True if both a and b are true, or both a and b are false
equivalent : Bool -> Bool -> Bool
```

Define a function to show a truth table for a logical expression.

```
truthTable : (Bool -> Bool -> Bool) -> List (Bool, Bool, Bool)
```

## Unit Test

```
import Html exposing (text)
import List exposing (map)
```

```
-- True if and only if a and b are true

and_ : Bool -> Bool -> Bool
and_ a b =
    -- your implementation here
    True

-- True if either a or b are true

or_ : Bool -> Bool -> Bool
or_ a b =
    -- your implementation here
    True

-- True either a or b are false

nand_ : Bool -> Bool -> Bool
nand_ a b =
    -- your implementation here
    True

-- True if and only if a and b are false

nor_ : Bool -> Bool -> Bool
nor_ a b =
    -- your implementation here
    True
```

```
-- True if a or b is true, but not if both are true

xor_ : Bool -> Bool -> Bool
xor_ a b =
    -- your implementation here
    True

-- True if a is false or b is true

implies : Bool -> Bool -> Bool
implies a b =
    -- your implementation here
    True

-- True if both a and b are true, or both a and b are false

equivalent : Bool -> Bool -> Bool
equivalent a b =
    -- your implementation here
    True

truthTable : (Bool -> Bool -> Bool) -> List ( Bool, Bool, Bool )
truthTable f =
    -- your implementation goes here
    List.repeat 4 ( True, True, True )

main =
    text
        (if (test) then
            "Your implementation passed all tests.")
```

```
        else
            "Your implementation failed at least one test."
    )

test : Bool
test =
    List.all ((==) True)
        [ truthTable and_
            == [ ( True, True, True )
                , ( True, False, False )
                , ( False, True, False )
                , ( False, False, False )
              ]
        , truthTable or_
            == [ ( True, True, True )
                , ( True, False, True )
                , ( False, True, True )
                , ( False, False, False )
              ]
        , truthTable nand_
            == [ ( True, True, False )
                , ( True, False, True )
                , ( False, True, True )
                , ( False, False, True )
              ]
        , truthTable nor_
            == [ ( True, True, False )
                , ( True, False, False )
                , ( False, True, False )
                , ( False, False, True )
              ]
        , truthTable xor_
            == [ ( True, True, False )
                , ( True, False, True )
                , ( False, True, True )
                , ( False, False, False )
              ]
        , truthTable implies
            == [ ( True, True, True )
```

```
      , ( True, False, False )
      , ( False, True, True )
      , ( False, False, True )
    ]
  , truthTable equivalent
    == [ ( True, True, True )
        , ( True, False, False )
        , ( False, True, False )
        , ( False, False, True )
      ]
  , truthTable (\a b -> (and_ a (or_ a b)))
    == [ ( True, True, True )
        , ( True, False, True )
        , ( False, True, False )
        , ( False, False, False )
      ]
]
```

## Solutions

### [Solutions](#)

## Problem 47

Create custom infix operators for the logical functions from Problem 46. Use the following operator tokens to replace the functions in [Problem 46](#).

```
.& = and_  
.| = or_  
/& = nand_  
/| = nor_  
*| for xor_  
.^ for implies  
.= for equivalent
```

Precedence and associativity of custom operators can be set with the keywords `infixl`, `infix`, and `infixr`. See <http://elm-lang.org/blog/announce/0.10#infix-operators>.

## Example

```
truthTable (\a b -> a .& ( a .| b)) ==  
  [ (True, True, True)  
    , (True, False, True)  
    , (False, True, False)  
    , (False, False, False)  
  ]  
  {--  
a=True  b=True  True  
a=True  b=False True  
a=False b=True  False  
a=False b=False False  
--}
```

## Unit Test

```
import Html exposing (text)
```

```
import List exposing (map)

truthTable : (Bool -> Bool -> Bool) -> List (Bool, Bool, Bool)
truthTable f =
    -- your implementation goes here
    List.repeat 4 (True, True, f True True)

main =
    text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all ((==) True)
        [ truthTable (.&) ==
            [ (True, True, True)
            , (True, False, False)
            , (False, True, False)
            , (False, False, False)
            ]
        , truthTable (.|) ==
            [ (True, True, True)
            , (True, False, True)
            , (False, True, True)
            , (False, False, False)
            ]
        , truthTable (/&) ==
            [ (True, True, False)
            , (True, False, True)
            , (False, True, True)
            , (False, False, True)
            ]
        , truthTable (/|) ==
```

```
[ (True, True, False)
, (True, False, False)
, (False, True, False)
, (False, False, True)
]
, truthTable (*|) ==
  [ (True, True, False)
  , (True, False, True)
  , (False, True, True)
  , (False, False, False)
  ]
, truthTable (.^ ) ==
  [ (True, True, True)
  , (True, False, False)
  , (False, True, True)
  , (False, False, True)
  ]
, truthTable (.=) ==
  [ (True, True, True)
  , (True, False, False)
  , (False, True, False)
  , (False, False, True)
  ]
, truthTable (\a b -> ( a .& ( a .| b))) ==
  [ (True, True, True)
  , (True, False, True)
  , (False, True, False)
  , (False, False, False)
  ]
]
```

## Solutions

[Solutions](#)



## Problem 49

The [Gray Code](#) is a binary numbering system used in error correction system. It can be generated for different bit sizes using a *reflecting* pattern. To generate a two-bit code, take the single bit Gray code 0, 1. Write it forwards, then backwards: 0, 1, 1, 0. Prepend 0s to the first half and 1s to the second half: 00, 01, 11, 10. To generate a 3-bit code, write 00, 01, 11, 10, 10, 11, 01, 00 to obtain: 000, 001, 011, 010, 110, 111, 101, 100.

## Example

```
grayCode 3 == [[0,0,0], [0,0,1], [0,1,1], [0,1,0], [1,1,0], [1,1,1], [1,0,1], [1,0,0]]
```

## Unit Test

```
import Html
import List

-- generate an array of integers encoded with Gray's binary code
grayCodes : Int -> List (List Int)
grayCodes numBits =
    let
        n =
            max 0 numBits
    in
        case n of
            0 ->
                []
            1 ->
                [ [ 0 ], [ 1 ] ]
```

```

    _ ->
        let
            n =
                abs numBits

            xs =
                grayCodes (n - 1)
        in
            (List.map ((::) 0) xs) ++ (List.map ((::) 1)
(List.reverse xs))

main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all ((==) True)
        [ grayCodes 1 == [ [ 0 ], [ 1 ] ]
        , grayCodes 2 == [ [ 0, 0 ], [ 0, 1 ], [ 1, 1 ], [ 1, 0
] ]
        , grayCodes 3 == [ [ 0, 0, 0 ], [ 0, 0, 1 ], [ 0, 1, 1 ]
, [ 0, 1, 0 ], [ 1, 1, 0 ], [ 1, 1, 1 ], [ 1, 0, 1 ], [ 1, 0, 0
] ]
        , grayCodes 4 == [ [ 0, 0, 0, 0 ], [ 0, 0, 0, 1 ], [ 0, 0
, 1, 1 ], [ 0, 0, 1, 0 ], [ 0, 1, 1, 0 ], [ 0, 1, 1, 1 ], [ 0, 1
, 0, 1 ], [ 0, 1, 0, 0 ], [ 1, 1, 0, 0 ], [ 1, 1, 0, 1 ], [ 1, 1
, 1, 1 ], [ 1, 1, 1, 0 ], [ 1, 0, 1, 0 ], [ 1, 0, 1, 1 ], [ 1, 0
, 0, 1 ], [ 1, 0, 0, 0 ] ]
        , testGray 4 (grayCodes 4)
        , testGray 5 (grayCodes 5)
        , testGray 8 (grayCodes 8)
        , testGray 13 (grayCodes 13)

```

```

    ]

getDeltas : List (List Int) -> List Int
getDeltas xs =
    case xs of
        [] ->
            [ 0 ]

        [ g1 ] ->
            [ 0 ]

        [ g1, g2 ] ->
            [ abs ((List.sum g1) - (List.sum g2)) ]

        g1 :: g2 :: gs ->
            abs ((List.sum g1) - (List.sum g2)) :: getDeltas (g2
:: gs)

-- In the Gray bit code the sum of the bits of two neighboring n
umbers
-- is always 1

testGray : Int -> List (List Int) -> Bool
testGray bits xs =
    let
        highest =
            Maybe.withDefault [ -5 ] <|
                List.head <|
                    List.reverse xs

        one =
            Maybe.withDefault [ -5 ] <|
                List.head <|
                    Maybe.withDefault [ [ -5 ] ] <|
                        List.tail xs

```

```
zero =
  Maybe.withDefault [ -5 ] <|
    List.head xs

deltas =
  getDeltas xs

in
  List.all ((==) True)
    [ List.sum highest == 1
    , List.sum one == 1
    , List.length xs == 2 ^ bits
    , List.sum zero == 0
    ]
    && List.all ((==) 1) deltas
```

## Hints

1. Use the reflecting nature of the Gray code to recursively solve for the answer.

## Solutions

[Solutions](#)

## Problem 50a - Huffman Encoding

Huffman coding uses variable bit length codes to efficiently encode data by giving the frequently found values short codes and rarely used values longer codes. The first step in Huffman encoding is to determine the frequency of every value in the input data.

### Example

```
freqs [1, 20, 20, 3, 3, 3, 3] == [(1, 1), (2, 20), (4, 3)]
```

### Unit Test

```
import Html
import List
import String

freqs : List comparable -> List ( Int, comparable )
freqs list =
  -- your implementation here
  []

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
```

```

List.all ((==) True)
  [ freqs [] == []
    , sortFreqs (freqs [ 20, 3, 20, 3, 1, 3, 3 ]) == [ ( 1, 1
), ( 2, 20 ), ( 4, 3 ) ]
    , sortFreqs (freqs [ 20, 3, 20, 3, 1, 3, 3 ]) == [ ( 1, 1
), ( 2, 20 ), ( 4, 3 ) ]
    , sortFreqs (freqs [ 3, 3, -20, 1, 3, 3, -20 ]) == [ ( 1
, 1 ), ( 2, -20 ), ( 4, 3 ) ]
    , (List.head <| List.reverse <| sortFreqs <| freqs <| to
Chars "hello world") == Just ( 3, 'l' )
    , sumFreqs freqs (toChars "hello world") == True
    , sumFreqs freqs (toChars "Now isthetimeforallgoodmentoc
ometothe...") == True
    , sumFreqs freqs (toChars "Now is the time for all good
men to come to the...") == True
    , sumFreqs freqs (toChars "El pingüino frío añoró") == T
rue
    , sumFreqs freqs (toChars "Да, но фальшивый экземпляр!")
== True
    , sumFreqs freqs (toChars "ἄγγελον ἀθανάτων ἐριούνιον, ὃ
ν τέκε Μαῖα") == True
  ]

```

```

sumFreqs : (List a -> List ( Int, a )) -> List a -> Bool
sumFreqs f list =
  List.sum (List.map fst (f list)) == List.length list

```

```

sortFreqs : List ( Int, a ) -> List ( Int, a )
sortFreqs list =
  List.sortBy (\( l, v ) -> l) list

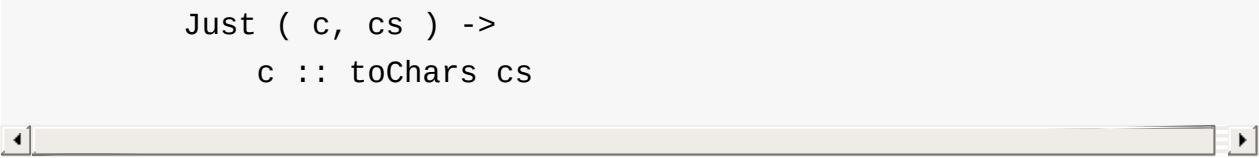
```

```

toChars : String -> List Char
toChars s =
  case String.uncons s of
    Nothing ->
      []

```

```
Just ( c, cs ) ->  
    c :: toChars cs
```



## Hints

1. Look at [Problem 9](#) and [Problem 10](#). Can you use or modify those solutions to this problem? Note we are assuming the type of the input list will be a `comparable` .

## Solutions

[Solutions](#)

## Problem 50b - Huffman Encoding

Huffman coding uses variable bit length codes to efficiently encode data. In [Problem 50a](#) we find the frequencies of each value of our input data. Now use that to build the Huffman codes. See the steps to build a Huffman code from the frequency data [here](#).

You'll need a binary tree to build the codes. You should complete [Problems 54 through 61](#) before attempting this problem.

### Example

```
huffman [('a', 45), ('b', 13), ('c', 12), ('d', 16), ('e', 9), ('f', 5)] ==  
[('a', "0"), ('b', "101"), ('c', "100"), ('d', "111"), ('e', "1101"), ('f',  
"1100")]
```

### Unit Test

```
import Html  
import List  
import Set  
import String  
  
type alias Freq = (Char, Int)  
  
huffman : List Freq -> List (Char, String)  
huffman list =  
    -- your implementation goes here  
    []  
  
main =
```



```

Html.text
  (if (test) then
    "Your implementation passed all tests."
  else
    "Your implementation failed at least one test."
  )

test : Bool
test =
  let
    codes = huffman [('a',45),('b',13),('c',12),('d',16),('e',9),('f',5)]
  in
    List.all ((==) True)
      [ assertCodeLength codes 'a' 1
        , assertCodeLength codes 'b' 3
        , assertCodeLength codes 'c' 3
        , assertCodeLength codes 'd' 3
        , assertCodeLength codes 'e' 4
        , assertCodeLength codes 'f' 4
        , List.length codes == 6
        , 6 == List.length (Set.toList (Set.fromList codes))
      ]

assertCodeLength : List (Char, String) -> Char -> Int -> Bool
assertCodeLength codes c n =
  case codes of
    [] -> False

    (ch, s) :: cs ->
      if ch == c then
        n == String.length s
      else
        assertCodeLength cs c n

```

## Solutions

Solutions

## Problem 54

Define a data type to represent a binary tree.

```
type Tree a =  
  -- your definition goes here
```

Solution

## Problem 55

In a balanced binary tree the difference between the number of nodes in the left and right subtrees of every node is less than one.

Write a function to construct a balanced binary tree for a given number of nodes. Put the letter 'x' as data value for all nodes of the tree.

## Example

```
balancedTree 3 ==  
  (Node 'x'  
    (Node 'x' (Empty) (Empty))  
    (Node 'x' (Empty) (Empty)))
```

## Unit Test

```
import Html  
import List  
import String  
  
type Tree a  
  = Empty  
  | Node a (Tree a) (Tree a)  
  
balancedTree : Int -> Tree Char  
balancedTree n =  
  -- your implementation here  
  Empty  
  
main =  
  Html.text
```

```

    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ count Empty == 0
    , count (Node 'x' (Empty) (Empty)) == 1
    , count (Node 'x' (Node 'x' (Empty) (Empty)) (Node 'x' (
Empty) (Empty))) == 3
    , count (Node 'x'
              (Node 'x' (Node 'x' (Empty) (Empty)) (Node '
x' (Empty) (Empty)))
              (Node 'x' (Node 'x' (Empty) (Empty)) (Node '
x' (Empty) (Empty)))) == 7
    , balancedTree 0 == Empty
    , balancedTree -1 == Empty
    , balancedTree 1 == (Node 'x' (Empty) (Empty))
    , balancedTree 3 == (Node 'x' (Node 'x' (Empty) (Empty))
(Node 'x' (Empty) (Empty)))
    , balancedTree 7 == (Node 'x'
                        (Node 'x' (Node 'x' (Empty) (Emp
ty))) (Node 'x' (Empty) (Empty)))
                        (Node 'x' (Node 'x' (Empty) (Emp
ty))) (Node 'x' (Empty) (Empty)))
    , testBalance (balancedTree 3) 3 == True
    , testBalance (balancedTree 4) 4 == True
    , testBalance (balancedTree 5) 5 == True
    , testBalance (balancedTree 7) 7 == True
    , testBalance (balancedTree 21) 21 == True
    , testBalance (balancedTree 31) 31 == True
    , testBalance (balancedTree 32) 32 == True
    , testBalance (balancedTree 33) 33 == True
    , testBalance (balancedTree 59) 59 == True
    , testBalance (balancedTree 64) 64 == True
    , testBalance (balancedTree 73) 73 == True

```

```
    , testBalance (balancedTree 5000) 5000 == True
  ]

testBalance : Tree a -> Int -> Bool
testBalance tree n =
  case tree of
    Empty ->
      n == 0

    Node node left right ->
      List.all ((==) True)
        [ (abs (count left) - count (right)) < 2
          , count tree == n
        ]

-- count number of Nodes in a Tree
count : Tree a -> Int
count tree =
  case tree of
    Empty -> 0

    Node n left right ->
      1 + (count left) + (count right)
```

## Solutions

[Solutions](#)

## Problem 56

We call a binary tree symmetric if you can draw a vertical line through the root node and the right subtree is the mirror image of the left subtree. Write a function to check if a binary tree is structurally symmetric (ignore the node values).

### Example

```
tree1 =  
  Node 1  
    (Node 2  
      (Node 3 Empty Empty)  
      (Node 4  
        Empty  
        (Node 5 Empty Empty)))  
    (Node 6  
      (Node 7 Empty Empty)  
      (Node 8  
        (Node 9 Empty Empty)  
        Empty)))  
  
isSymmetric tree1 == True
```

### Unit Test

```
import Html  
import List  
  
type Tree a  
  = Empty  
  | Node a (Tree a) (Tree a)
```

```
isSymmetric : Tree a -> Bool
isSymmetric tree =
    -- your implementation goes here
    False

main : Html.Html a
main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all ((==) True)
        [ isSymmetric tree1
        , isSymmetric (Node '1' Empty (Node '2' Empty Empty)) ==
False
        , isSymmetric (Node '1' (Node '2' Empty Empty) (Node '3'
Empty (Node '4' Empty Empty))) == False
        , isSymmetric Empty
        , isSymmetric (balancedTree 3)
        , isSymmetric (balancedTree 4) == False
        , isSymmetric (balancedTree 6) == False
        , isSymmetric (balancedTree 7)
        , isSymmetric (balancedTree 15)
        , isSymmetric (balancedTree 31)
        , isSymmetric (balancedTree 32) == False
        ]

tree1 =
    Node 1
        (Node 2
            (Node 3 Empty Empty)
            (Node 4
```



```

        Empty
        (Node 5 Empty Empty)))
(Node 6
  (Node 8
    (Node 9 Empty Empty)
    Empty)
  (Node 7 Empty Empty))

balancedTree : Int -> Tree Char
balancedTree n =
  List.foldl (addBalancedNode) Empty <| List.repeat n 'x'

addBalancedNode : a -> Tree a -> Tree a
addBalancedNode v tree =
  case tree of
    Empty ->
      Node v Empty Empty

    Node v_ left right ->
      if count left > count right then
        Node v_ left (addBalancedNode v right)
      else
        Node v_ (addBalancedNode v left) right

-- count number of Nodes in a Tree
count : Tree a -> Int
count tree =
  case tree of
    Empty -> 0

    Node n left right ->
      1 + (count left) + (count right)

```

## Hint

1. Look in the mirror (recursively).

# Solutions

Solutions

## Problem 57

Build a [binary search tree](#) from a list. Place lower values the right. By definition, duplicate values are omitted.

## Example

```
toBSTree [6, 2, 4, 20, 1, 11, 12, 14] ==  
  Node 6  
    (Node 2  
      (Node 1 Empty Empty)  
      (Node 4 Empty Empty))  
    (Node 20  
      (Node 11  
        Empty  
        (Node 12  
          Empty  
          (Node 14 Empty Empty)))  
      Empty)  
    Empty)
```

## Unit Test

```
import Html  
import List  
  
type Tree a  
  = Empty  
  | Node a (Tree a) (Tree a)  
  
toBSTree : List comparable -> Tree comparable  
toBSTree list =  
  -- your implementation goes here  
  Empty
```

```
main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ toBSTree [] == Empty
    , toBSTree [1] == Node 1 Empty Empty
    , toBSTree [1, 1, 1] == Node 1 Empty Empty
    , toBSTree (List.range 1 5) == Node 1 Empty (Node 2 Empty
(Node 3 Empty (Node 4 Empty (Node 5 Empty Empty))))
    , toBSTree (List.reverse (List.range 1 5)) == Node 5 (No
de 4 (Node 3 (Node 2 (Node 1 Empty Empty) Empty) Empty) Empty) E
mpty
    , toBSTree [6, 2, 4, 20, 1, 11, 12, 14, 6] ==
      Node 6
        (Node 2
          (Node 1 Empty Empty)
          (Node 4 Empty Empty))
        (Node 20
          (Node 11
            Empty
            (Node 12
              Empty
              (Node 14 Empty Empty))))
          Empty)
    ]
```

## Hints

1. The solution to [Problem 55](#) can be easily modified to solve this problem.

## Solutions

### [Solutions](#)

## Problem 61a

Count the leaves of a binary tree. The leaves are the nodes that have empty nodes in both their right and left subtrees.

### Example

```
tree = Tree 1 (Tree 2 Empty (Tree 4 Empty Empty))
      (Tree 2 Empty Empty)

countLeaves tree == 2
```

### Unit Test

```
import Html
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

countLeaves : Tree a -> Int
countLeaves tree =
  -- your implementation here
  0

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ countLeaves Empty == 0
    , countLeaves (Node 1 Empty Empty) == 1
    , countLeaves (Node 1 (Node 2 Empty Empty) Empty) == 1
    , countLeaves (Node 1 (Node 2 Empty Empty) (Node 0 Empty
Empty)) == 2
    , countLeaves (Node 1 (Node 2 Empty Empty) (Node 0 Empty
Empty)) == 2
    ]
```

## Hints

1. You should be pretty good at recursion by now.

## Solutions

[Solutions](#)



## Problem 61b

Extract the values of the leaves of a binary tree into a list.

### Example

```
tree = Tree 1 (Tree 2 Empty (Tree 4 Empty Empty))
      (Tree 2 Empty Empty)

getLeaves aTree == [4, 2]
```

### Unit Test

```
import Html
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

getLeaves : Tree comparable -> List comparable
getLeaves tree =
  -- your implementation here
  []

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )
```

```
test : Bool
test =
  List.all ((==) True)
    [ getLeaves Empty == []
      , getLeaves (Node 1 Empty Empty) == [1]
      , List.sort (getLeaves (Node 1 (Node 2 Empty Empty) Empty
    )) == [2]
      , List.sort (getLeaves (Node 1 (Node 2 Empty Empty) (Node
0 Empty Empty)))
        == [0, 2]
      , List.sort (getLeaves (Node 1 (Node 2 Empty Empty) (Node
0 Empty Empty)))
        == [0, 2]
      , List.sort (getLeaves (Node "A" (Node "B" Empty Empty) (
Node "C" Empty Empty)))
        == ["B", "C"]
    ]
```

## Hints

1. You don't need no stinking hints!

## Solutions

[Solutions](#)

## Problem 62a

Count internal nodes (those that have non-empty subtrees) of a binary tree.

### Example

```
tree = Tree 'a' (Tree 'b' Empty (Tree 'c' Empty Empty))
      (Tree 'd' Empty (Tree 'e' Empty Empty))

countInternals tree == 3
```

### Unit Test

```
import Html
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

countInternals : Tree a -> Int
countInternals tree =
  -- your implementation goes here
  0

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )
```

```
test : Bool
test =
  List.all ((==) True)
    [ countInternals Empty == 0
    , countInternals (Node 1 Empty Empty) == 0
    , countInternals (Node 1 (Node 2 Empty Empty) Empty) == 1

    , countInternals (Node 1 (Node 2 Empty Empty) (Node 0 Empty Empty)) == 1
    , countInternals (Node "A" (Node "B" Empty Empty) (Node "C" Empty Empty)) == 1
    , countInternals (Node 1 (Node 1 Empty (Node 4 Empty Empty))
                          (Node 1 (Node 2 Empty Empty) Empty)) == 3
    ]
```

## Hints

1. You don't need no stinking hints!

## Solutions

[Solutions](#)

## Problem 62b

Extract the internal nodes (those that have non-empty subtrees) of a binary tree into a list.

### Example

```
tree = Tree 'a' (Tree 'b' Empty (Tree 'c' Empty Empty))
        (Tree 'd' Empty (Tree 'e' Empty Empty))

getInternals tree == ['a', 'b', 'd']
```

### Unit Test

```
import Html
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

getInternals : Tree a -> List a
getInternals tree =
  -- your implemenation goes here
  []

main: Html.Html a
main =
  Html.text
    <| case test of
        0 ->
          "Your implementation passed all tests."
```

```

1 ->
    "Your implementation failed one test."

x ->
    "Your implementation failed " ++ (toString x) ++
    " tests."

test : Int
test =
    List.length
    <| List.filter ((==) False)
        [ getInternals Empty == []
          , getInternals (Node 1 Empty Empty) == []
          , List.sort (getInternals (Node 1 (Node 2 Empty Empty)
) Empty)) == [ 1 ]
          , List.sort (getInternals (Node 1 (Node 2 Empty Empty)
) (Node 0 Empty Empty))) == [ 1 ]
          , List.sort (getInternals (Node "A" (Node "B" Empty
Empty) (Node "C" Empty Empty))) == [ "A" ]
          , List.sort
              (getInternals
                  (Node 1
                      (Node 1 Empty (Node 4 Empty Empty))
                      (Node 1 (Node 2 Empty Empty) Empty)
                  )
              )
          == [1, 1, 1]
    ]

```

## Hints

1. You don't need no stinking hints!

## Solutions

## Solutions

## Problem 63

We define a complete binary tree with height  $H$  as a tree where:

- levels  $1, \dots, H-1$  contain the maximum number of nodes,
- and in level  $H$  all the nodes are "left-adjusted".

## Example

```
completeTree 4 'x' ==  
Tree 'x' (Tree 'x' (Tree 'x' Empty Empty) Empty) (Tree 'x' Empty  
Empty)
```

## Unit Test



```

import Html
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

completeTree : a -> Int -> Tree a
completeTree v n =
  -- your implementation here
  Empty

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ completeTree 'x' 0 == Empty
    , completeTree 'x' 1 == Node 'x' Empty Empty
    , completeTree 'x' 3 == (Node 'x' (Node 'x' Empty Empty)
      (Node 'x' Empty Empty))
    , completeTree 'x' 5 == (Node 'x'
      (Node 'x' (Node 'x' Empty Em
pty) (Node 'x' Empty Empty))
      (Node 'x' Empty Empty))
    ]

```

## Hints

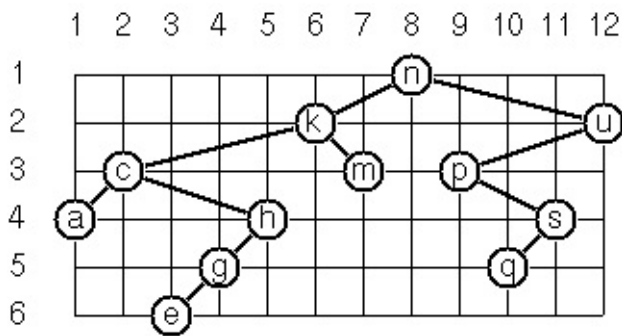
Heaps commonly use complete binary trees as data structures or addressing schemes. We can assign an address to each node in a complete binary tree by enumerating the nodes in level-order starting at the root with number 1. For every node  $x$  with address  $A$  the following holds: The address of  $x$ 's left and right successors are  $2 \cdot A$  and  $2 \cdot A + 1$ , respectively, if they exist. This fact can be used to elegantly construct a complete binary tree structure.

## Solutions

### Solutions

## Problem 64

To draw a tree a layout algorithm determines the position of each node in a rectangular grid. Several layout methods are conceivable, one of them is illustrated below:



In this layout strategy, the following two rules determine the position:

- $x(v)$  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.

Write a function to annotate each node of the tree with its position. To solve this problem you will need to traverse the tree. Problems [68a](#), [68b](#), and [68c](#) examine tree traversal.

## Example

```

tree64 =
  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)

layout tree64 ==
  Node ('n',(8,1)) (Node ('k',(6,2)) (Node ('c',(2,3)) ...

```

## Unit Test

```

import Html
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

tree64 =
  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)
Empty)
```

```
layout : Tree a -> Tree (a, (Int, Int))
```

```
layout tree =
```

```
-- your implementation here
```

```
Empty
```

```
main =
```

```
Html.text
```

```
  (if (test) then
```

```
    "Your implementation passed all tests."
```

```
  else
```

```
    "Your implementation failed at least one test."
```

```
  )
```

```
test : Bool
```

```
test =
```

```
  let
```

```
    t = layout tree64
```

```
  in
```

```
    List.all ((==) True)
```

```
      [ t == layout64
```

```
      ]
```

```

layout64 =
  Node ('n', (8, 1))
    (Node ('k', (6, 2))
      (Node ('c', (2, 3))
        (Node ('a', (1, 4)) Empty Empty)
        (Node ('h', (5, 4))
          (Node ('g', (4, 5))
            (Node ('e', (3, 6)) Empty Empty)
            Empty)
          Empty))
        (Node ('m', (7, 3)) Empty Empty))
      (Node ('u', (12, 2))
        (Node ('p', (9, 3))
          Empty
          (Node ('s', (11, 4))
            (Node ('q', (10, 5)) Empty Empty)
            Empty))
          Empty)
        Empty)
    )

```

## Hints

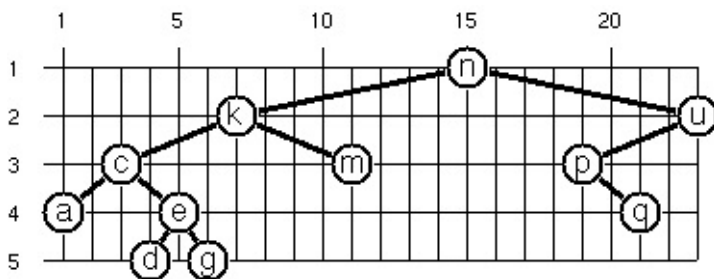
1. You can simplify the problem by breaking down into two or more steps, traversing once to set the X, once to set the Y. You can model `List.indexedMap` to create traversal functions with depth or in-order.
2. The problem can be solved in a single traversal. You'll probably need nested `let` clauses to get the in-order values from subtree traversals.

## Solutions

### Solutions

## Problem 65

The illustration below depicts an alternative layout method.



Find out the rules and write the

corresponding function. Hint: On a given level, the horizontal distance between neighboring nodes is constant. Write a function to annotate each node of the tree with its position.

To solve this problem you will need to traverse the tree. Problems [68a](#), [68b](#), and [68c](#) examine tree traversal.

## Example

```

tree65 =
  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)

layout65 tree65 ==
  Node ('n',(15,1))
    (Node ('k',(7,2))
      (Node ('c',(3,3)) ...

```

## Unit Test

```

import Html
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

tree65 =
  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)

```

```

{-| Given a Tree return a tree that adds x, y coordinates for each node

```

```

    to layout a graphic representation of the tree

```

```

-}

```

```

layout : Tree comparable -> Tree ( comparable, (Int, Int) )

```

```

layout tree =

```

```

    -- your implementation goes here

```

```

    Empty

```

```

main =

```

```

    Html.text

```

```

        (if (test) then

```

```

            "Your implementation passed all tests."

```

```

        else

```

```

            "Your implementation failed at least one test."

```

```

        )

```

```

test : Bool

```

```

test =

```

```

    let

```

```

        t = layout tree65

```

```

    in

```

```

        List.all ((==) True)

```

```

            [ t == layout65

```

```

            ]

```

```

layout65 =

```

```
Node ('n', (15, 1))
  (Node ('k', (7, 2))
    (Node ('c', (3, 3))
      (Node ('a', (1, 4)) Empty Empty)
      (Node ('e', (5, 4))
        (Node ('d', (4, 5)) Empty Empty)
        (Node ('g', (6, 5)) Empty Empty)))
      (Node ('m', (11, 3)) Empty Empty))
    (Node ('u', (23, 2))
      (Node ('p', (19, 3))
        Empty
        (Node ('q', (21, 4)) Empty Empty))
      Empty))
```

## Hints

1. Find out the rules and write the corresponding function. Hint: At a given depth, the horizontal distance between neighboring nodes is constant.
2. Need another hint? The horizontal distance is a function of the depth from the tree from that node and the depth of the left subtree.

## Solutions

[Solutions](#)

## Problem 67

Consider a string representation of a tree such as:

```
a(b(d,e),c(,f(g,)))
```

Write a function to generate a string representation of a tree.

## Example

```
tree =  
  Node 'x'  
    (Node 'y' Empty Empty)  
    (Node 'a'  
      Empty  
      (Node 'b' Empty Empty))  
  
s = "x(y,a(,b))"  
  
toString tree == "x(y,a(,b))"
```

## Unit test

```
import Html  
import List  
import String  
  
type Tree a  
  = Empty  
  | Node a (Tree a) (Tree a)  
  
treeToString : Tree a -> String
```

```
treeToString tree =
  -- your implementation goes here
  ""

noQuotes : Char -> Bool
noQuotes c =
  (c /= '\\' ) && (c /= '\"')

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ treeToString tChar == "x(y,a(,b))"
    , treeToString (Node 'z' Empty tChar) == "z(,x(y,a(,b)))"
    , treeToString (Node 'z' tChar Empty) == "z(x(y,a(,b)),)"
    , treeToString tString == "x(y,a(,b))"
    , treeToString tInt == "8(9,1(,2))"
    , treeToString (Node 3 (Node 4 tInt Empty) Empty)
      == "3(4(8(9,1(,2))),,)"
    ]

tChar =
  Node 'x'
    (Node 'y' Empty Empty)
    (Node 'a'
      Empty
      (Node 'b' Empty Empty))

tString =
  Node "x"
```

```
(Node "y" Empty Empty)
(Node "a"
  Empty
  (Node "b" Empty Empty))

tInt =
  Node 8
    (Node 9 Empty Empty)
    (Node 1
      Empty
      (Node 2 Empty Empty))
```

## Hints

1. Solve the problem for just one Tree type first, `Tree Int` for example.

## Solutions

[Solutions](#)

## Problem 67b

Consider a string representation of a tree with a single character naming each node such as:

```
Write a function to generate a tree from its string representation created in [Problem 67a](p67a.md).
```

```
##Example
```elm
toTree "x(y,a(,b))" ==
  Node 'x'
    (Node 'y' Empty Empty)
    (Node 'a'
      Empty
      (Node 'b' Empty Empty))
```

## Unit test

```
import Html
import List
import String

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

toTree : String -> Tree Char
toTree tree =
  -- your implementation goes here
  Empty

main =
```

```
Html.text
  (if (test) then
    "Your implementation passed all tests."
  else
    "Your implementation failed at least one test."
  )

test : Bool
test =
  List.all ((==) True)
    [ toTree "" == Empty
    , toTree s1 == t1
    , toTree s2 == t2
    , toTree s3 == t3
    ]

s1 = "a(b(d,e),c(,f(g,)))"
t1 =
  Node 'a'
    (Node 'b'
      (Node 'd' Empty Empty)
      (Node 'e' Empty Empty))
    (Node 'c'
      Empty
      (Node 'f'
        (Node 'g' Empty Empty)
        Empty))

s2 = "x(y,a(,b))"
t2 =
  Node 'x'
    (Node 'y' Empty Empty)
    (Node 'a'
      Empty
      (Node 'b' Empty Empty))
```

```
s3 = "3(4(8(9,1(,2)),),,)"
t3 = Node '3' (Node '4' t4 Empty) Empty
t4 =
  Node '8'
    (Node '9' Empty Empty)
    (Node '1'
      Empty
      (Node '2' Empty Empty))
```

## Hints

1. Solve the problem for just one Tree type first, `Tree Int` for example.

## Solutions

[Solutions](#)



## Problem 68a

There are three commonly used traversal forms, pre-order, in-order and post-order.

- Pre-order traversal of a binary tree displays the data value of the node, then traverses the left subtree then traverses the right subtree. For example, pre-order traversal of the tree from Problem 67 gives us `a, b, d, e, c, f, g`.
- In-order traversal traverses the left subtree, then shows the data value, then traverses the right subtree.
- Post-order traverses the left subtree, then the right subtree, then shows the data value.

Write a function to traverse a tree and return the values in a list in pre-order.

## Example

```
tree67 =
  Node 'a'
    (Node 'b'
      (Node 'd' Empty Empty)
      (Node 'e' Empty Empty))
    (Node 'c'
      Empty
      (Node 'f'
        (Node 'g' Empty Empty)
        Empty))

preorder tree67 == ['a', 'b', 'd', 'e', 'c', 'f', 'g']
```

## Unit Test

```
import Html
```

```
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

preorder : Tree a -> List a
preorder tree =
  -- your implementation goes here
  []

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ preorder Empty == []
    , preorder tree67 == ['a', 'b', 'd', 'e', 'c', 'f', 'g']
    , preorder tree68 == (List.range 1 9)
    , preorder tree == [6, 2, 1, 4, 3, 5, 7]
    ]

tree67 =
  Node 'a'
    (Node 'b'
      (Node 'd' Empty Empty)
      (Node 'e' Empty Empty))
    (Node 'c'
      Empty
      (Node 'f'
```

```
        (Node 'g' Empty Empty)
      Empty))

tree68 =
  Node 1
    (Node 2
      (Node 3 Empty Empty)
      (Node 4 Empty Empty))
    (Node 5
      Empty
      (Node 6
        (Node 7
          Empty
          (Node 8
            (Node 9 Empty Empty)
            Empty))
        Empty))
    Empty))

tree =
  Node 6
    (Node 2
      (Node 1 Empty Empty)
      (Node 4
        (Node 3 Empty Empty)
        (Node 5 Empty Empty)))
    (Node 7 Empty Empty)
```

## Solution

## Problem 68b

Write a function to traverse a tree and return the values in a list in in-order.

### Example

```
tree67 =  
  Node 'a'  
    (Node 'b'  
      (Node 'd' Empty Empty)  
      (Node 'e' Empty Empty))  
    (Node 'c'  
      Empty  
      (Node 'f'  
        (Node 'g' Empty Empty)  
        Empty))  
inorder tree67 == ['d', 'b', 'e', 'a', 'c', 'g', 'f']
```

### Unit Test

```
\import Html  
import List  
  
type Tree a  
  = Empty  
  | Node a (Tree a) (Tree a)  
  
inorder : Tree a -> List a  
inorder tree =  
  -- your implementation goes here  
  []
```

```
main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ inorder Empty == []
    , inorder tree == (List.range 1 7)
    , inorder tree67 == ['d', 'b', 'e', 'a', 'c', 'g', 'f']
    , inorder tree68 == [3, 2, 4, 1, 5, 7, 9, 8, 6]
    ]

tree67 =
  Node 'a'
    (Node 'b'
      (Node 'd' Empty Empty)
      (Node 'e' Empty Empty))
    (Node 'c'
      Empty
      (Node 'f'
        (Node 'g' Empty Empty)
        Empty))

tree68 =
  Node 1
    (Node 2
      (Node 3 Empty Empty)
      (Node 4 Empty Empty))
    (Node 5
      Empty
      (Node 6
        (Node 7
```

```
Empty
(Node 8
  (Node 9 Empty Empty)
  Empty))
Empty))

tree =
  Node 6
    (Node 2
      (Node 1 Empty Empty)
      (Node 4
        (Node 3 Empty Empty)
        (Node 5 Empty Empty)))
    (Node 7 Empty Empty)
```

## Solution

## Problem 68c

Write a function to traverse a tree and return the values in a list in post-order.

### Example

```
tree67 =
  Node 'a'
    (Node 'b'
      (Node 'd' Empty Empty)
      (Node 'e' Empty Empty))
    (Node 'c'
      Empty
      (Node 'f'
        (Node 'g' Empty Empty)
        Empty))

postorder tree67 == ['d', 'e', 'b', 'g', 'f', 'c', 'a']
```

### Unit Test

```
import Html
import List

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

postorder : Tree a -> List a
postorder tree =
  -- your implementation goes here
  []
```

```
main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ postorder Empty == []
    , postorder tree67 == ['d', 'e', 'b', 'g', 'f', 'c', 'a']
    , postorder tree68 == [3, 4, 2, 9, 8, 7, 6, 5, 1]
    , postorder tree == [1, 3, 5, 4, 2, 7, 6]
    ]

tree67 =
  Node 'a'
    (Node 'b'
      (Node 'd' Empty Empty)
      (Node 'e' Empty Empty))
    (Node 'c'
      Empty
      (Node 'f'
        (Node 'g' Empty Empty)
        Empty))

tree68 =
  Node 1
    (Node 2
      (Node 3 Empty Empty)
      (Node 4 Empty Empty))
    (Node 5
      Empty
      (Node 6
        (Node 7
          Empty
```



```
                (Node 8
                  (Node 9 Empty Empty)
                  Empty))
            Empty))

tree =
  Node 6
    (Node 2
      (Node 1 Empty Empty)
      (Node 4
        (Node 3 Empty Empty)
        (Node 5 Empty Empty)))
    (Node 7 Empty Empty)
```

Solution

## Problem 69

Using the preorder sequence of values and dots (.) to represent empty subtrees we can unambiguously represent a binary tree.

- a) Write a function to create the dot-string representation of a tree.
- b) Write a function to build a tree from a dot-string.

## Example

```
tree69 = "abd..e..c.fg..."
tree69 == treeToDot <| dotToTree tree69
```

## Unit Test

```
import Html
import List
import Maybe
import String

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

dotToTree : String -> (Tree Char, String)
dotToTree s =
  -- your implementation here
  (Empty, "Fail")

treeToDot : Tree Char -> String
treeToDot tree =
```

```
-- your implementation here
"fail"

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ tree67 == Tuple.first (dotToTree dot67)
    , dot67 == treeToDot tree67
    , tree69a == Tuple.first (dotToTree dot69a)
    , dot69a == treeToDot tree69a
    , tree69b == Tuple.first (dotToTree dot69b)
    , dot69b == treeToDot tree69b
    ]

dot67 = "abd..e..c.fg..."

tree67 =
  Node 'a'
    (Node 'b'
      (Node 'd' Empty Empty)
      (Node 'e' Empty Empty))
    (Node 'c'
      Empty
      (Node 'f'
        (Node 'g' Empty Empty)
        Empty))

dot69a = "123..4..5.67.89...."
```

```
tree69a =
  Node '1'
    (Node '2'
      (Node '3' Empty Empty)
      (Node '4' Empty Empty))
    (Node '5'
      Empty
      (Node '6'
        (Node '7'
          Empty
          (Node '8'
            (Node '9' Empty Empty)
            Empty))
        Empty))

dot69b = "621..43..5..7.."

tree69b =
  Node '6'
    (Node '2'
      (Node '1' Empty Empty)
      (Node '4'
        (Node '3' Empty Empty)
        (Node '5' Empty Empty)))
    (Node '7' Empty Empty)
```

## Hints

1. `String.uncons` and `String.fromChar` may come in handy.
2. To convert from string to tree recurse in preorder with a function that returns two values, the tree value and the unused portion of the string.

## Solution

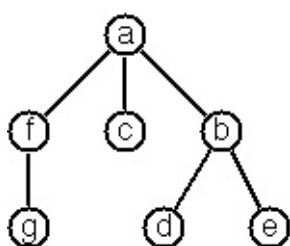
[Solution](#)



## Problem 70a

A multiway tree has a root element and a set of successors which are multiway trees themselves. A multiway tree is never empty, although it's list of children may be an empty list.

- Define type to represent multiway trees.
- Create a tree instance to represent the tree in the diagram below.
- Write a function to count all nodes of the tree.



## Unit Test

```
import Html
import List

type MTree a =
    -- your definition goes here

mtree70 =
    -- instantiate the multiway tree illutrated above

count : MTree a -> Int
    -- your definition goes here

main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all ((==) True)
        [ count mtree70 == 7
        ]
```

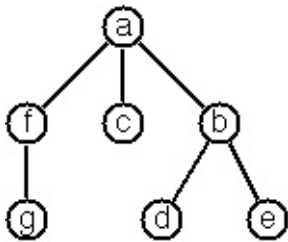
## Solution

[Solution](#)

## Problem 70b

Generate a string representation of a multiway tree in depth-first order. Insert the special character `^` whenever the move is a backtrack to the previous level.

By this rule, the tree below is represented as: `afg^^c^bd^e^^^`



- Write a function to convert a multiway tree to a string representation.
- Write a function to convert a string representation to a multiway tree.

## Example

```

type MTree a = MNode a (List (MTree a))

mtree70 = MNode 'a'
          [ MNode 'f' [ MNode 'g' [] ]
            , MNode 'c' []
            , MNode 'b' [ MNode 'd' [], MNode 'e' [] ]
          ]

s70 = "afg^^c^bd^e^^^"

s70 == treeToString <| stringToTree s70

```

## Unit Test

```

import Html
import List
import Maybe

```



```
import String

type MTree a = MNode a (List (MTree a))

stringToTree : String -> MTree Char
stringToTree s =
    -- your implementation goes here
    MNode 'X' []

treeToString : MTree Char -> String
treeToString tree =
    "Your implementation goes here"

main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all ((==) True)
        [ mtree70 == stringToTree string70
        , string70 == treeToString mtree70
        , aT == stringToTree aS
        , aS == treeToString aT
        ]

aS = "af^c^b^^"
aT = MNode 'a'
    [ MNode 'f' []
    , MNode 'c' []
    , MNode 'b' []
```

```
    ]

string70 = "afg^^c^bd^e^^^"
mtree70 = MNode 'a'
    [ MNode 'f' [ MNode 'g' [] ]
    , MNode 'c' []
    , MNode 'b' [ MNode 'd' [], MNode 'e' [] ]
    ]
```

## Hints

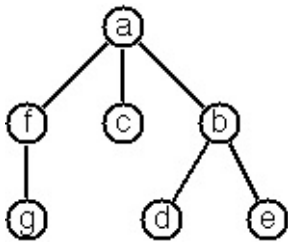
1. This problem is similar to [Problem 69](#).
2. Traverse the tree in pre-order with a function that returns two values, the tree value and the unused portion of the string.
3. Treating the root node differently than the rest of the tree may result in simpler functions.
4. `List.intersperse`, `List.fromChar`, `List.cons` and `List.uncons` may come in handy

## Solution

[Solution](#)

## Problem 71

The internal path length of a multiway tree is the total sum of the path lengths from the root to all nodes of the tree. The tree shown below has an internal path length of 9.



Write a function to determine the internal path length of a tree

```
mtree71 = Node 'a' [Node 'f' [Node 'g'], Node 'c'[], Node 'b'[Node 'd' [], Node 'e' []]]
```

```
internalPathLength mtree71 == 9
```

## Unit Test

```
import Html
import List
```

```
type MTree a = MNode a (List (MTree a))
```

```
mtree70 = MNode 'a'
  [ MNode 'f' [ MNode 'g' [] ]
  , MNode 'c' []
  , MNode 'b' [ MNode 'd' [], MNode 'e' [] ]
  ]
```

```
internalPathLength : MTree a -> Int
```

```

internalPathLength mtree =
    -- your implementation goes here
    0

main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all ((==) True)
        [ internalPathLength (MNode 10 []) == 0
        , internalPathLength mtree70 == 9
        , internalPathLength test3 == 10
        , internalPathLength test4 == 4
        , internalPathLength test5 == 30
        ]

test3 = MNode '1' [ MNode '2' [ MNode '3' [ MNode '4' [ MNode '5'
    ' [] ] ] ] ] ]
test4 = MNode 'a' [ MNode 'f' [], MNode 'g' [], MNode 'g' [], MN
ode 'g' [] ]
test5 = MNode '1' [ MNode '2' [ MNode '3' [ MNode '4' [ test4 ]
] ] ]

```

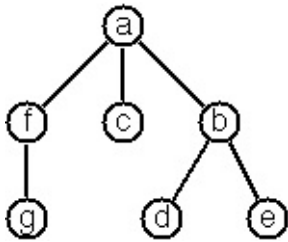
## Solution

[Solution](#)

## Problem 72

Collect the nodes of a multiway tree in depth-first order.

### Example



```
toList tree70 == ['g', 'f', 'c', 'd', 'e', 'b', 'a']
```

### Unit Test

```
import Html
import List

type MTree a
  = MNode a (List (MTree a))

toList : MTree a -> List a
toList (MNode v list) =
  -- your implementation goes here
  []

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
```

```

        "Your implementation failed at least one test."
    )

test : Bool
test =
    List.all ((==) True)
        [ toList (MNode 1 []) == [ 1 ]
          , toList mtree70 == [ 'g', 'f', 'c', 'd', 'e', 'b', 'a'
                                ]
          , toList test3 == [ '5', '4', '3', '2', '1' ]
          , toList test4 == [ 'f', 'g', 'h', 'i', 'a' ]
          , toList test5 == [ 'f', 'g', 'h', 'i', 'a', '4', '3', '2
                                ', '1' ]
                                ]

mtree70 =
    MNode 'a'
        [ MNode 'f' [ MNode 'g' [] ]
          , MNode 'c' []
          , MNode 'b' [ MNode 'd' [], MNode 'e' [] ]
        ]

test3 =
    MNode '1' [ MNode '2' [ MNode '3' [ MNode '4' [ MNode '5' []
  ] ] ] ]

test4 =
    MNode 'a' [ MNode 'f' [], MNode 'g' [], MNode 'h' [], MNode
                'i' [] ]

test5 =
    MNode '1' [ MNode '2' [ MNode '3' [ MNode '4' [ test4 ] ] ]
                ]

```

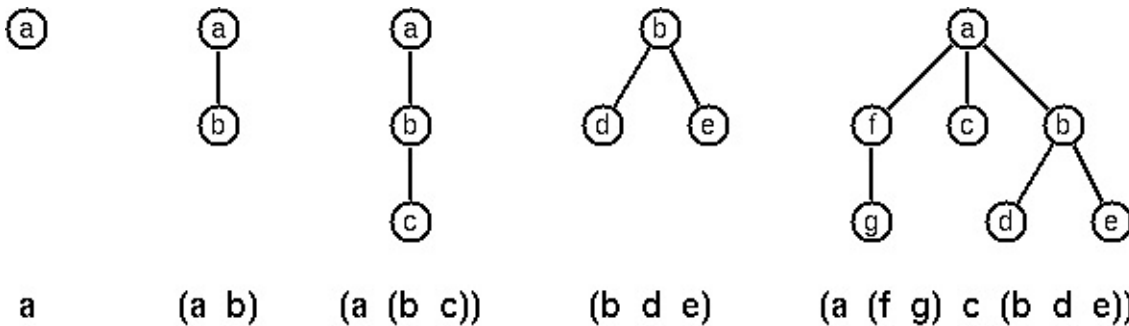
## Hint

1. Depth-first again?
2. List.concatMap could make the solution very concise.

## Solution

[Solution](#)

## Problem 73



a) Write a function of represent a multiway tree as a String, using the "Lispy" convention illustrated above.

b) Build an Tree from a "Lispy" string.

```
import Html exposing (text)

type Tree a = Node a (List (Tree a))

tree1 = Node 'a' []
tree2 = Node 'a' [Node 'b' []]
tree3 = Node 'a' [Node 'b' [Node 'c' []]]
tree4 = Node 'b' [Node 'd' [], Node 'e' []]
tree5 =
  Node 'a'
    [ Node 'f'
      [ Node 'g' []
      , Node 'c' []
      , Node 'b'
        [ Node 'd' []
        , Node 'e' []
        ]
      ]
    ]
s5 = "(a(f(g)cb(de)))"

lispify : Tree a -> String
lispify t =
```



```
-- your implementation here
""

delispify : String -> Tree Char
delispify s =
  -- your implementation here
  Node 'x' []

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ tree4 == delispify "(b(d(d)))"
    , lispify tree4 == "(b(d(d)))"
    , tree5 == delispify s5
    , lispify tree5 == s5
    ]
```

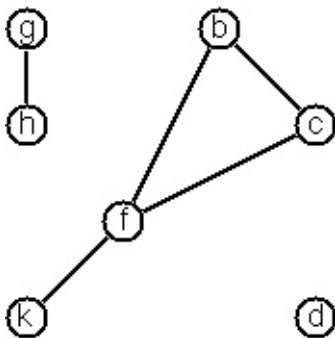
## Result

```
['g', 'f', 'c', 'd', 'e', 'b', 'a']
```

## Problem 80a

Convert from a graph from graph-term to adjacency-list representation. (See [Graphs](#) for details on these formats.)

### Example



```
graph80 = ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ],
            [ ('b','c'), ('b','f'), ('c','f'), ('f','k'), ('g','h') ] )
```

```
adjList80 =
  [ ('b',['c','f'])
  , ('c',['b','f'])
  , ('d',[])
  , ('f',['b','c','k'])
  , ('g',['h'])
  , ('h',['g'])
  , ('k',['f'])
  ]
```

```
adjList80 == graphToAdjList graph80
```

### Unit Test

```
import Html
import List
```

```
import Set

type alias Edge comparable = (comparable, comparable)
type alias AdjList comparable = (List((comparable, List comparable)))

type alias Graph comparable = (List comparable, List (Edge comparable))

graphToAdjList : Graph comparable -> AdjList comparable
graphToAdjList (nodes, edges) =
  -- your implementation goes here
  []

main =
  Html.text
    (if (test) then
      "Your implementation passed all tests."
    else
      "Your implementation failed at least one test."
    )

test : Bool
test =
  List.all ((==) True)
    [ adjList80 == graphToAdjList graph80
    , [] == graphToAdjList ([], [])
    ]

graph80 = ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ],
             [ ('b','c'), ('b','f'), ('c','f'), ('f','k'), ('g','h') ] )

adjList80 =
  [ ('b',['c','f'])
```

```
, ('c', ['b', 'f'])  
, ('d', [])  
, ('f', ['b', 'c', 'k'])  
, ('g', ['h'])  
, ('h', ['g'])  
, ('k', ['f'])  
]
```

## Hints

1. `Set.fromList` and `Set.toList` may come in handy.

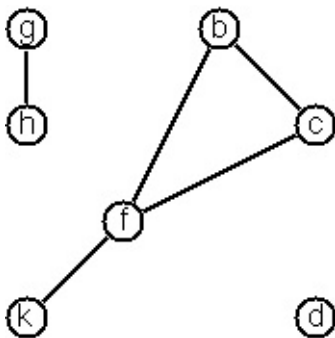
## Solution

[Solution](#)

## Problem 80b

Convert from a graph from adjacency-list to graph-term representation. (See [Graphs](#) for details on these formats.)

### Example



```
graph80 = ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ],
            [ ('b','c'), ('b','f'), ('c','f'), ('f','k'), ('g','h') ] )
```

```
adjList80 =
  [ ('b',['c','f'])
  , ('c',['b','f'])
  , ('d',[])
  , ('f',['b','c','k'])
  , ('g',['h'])
  , ('h',['g'])
  , ('k',['f'])
  ]
```

```
graph80 == adjListToGraph adjList80
```

### Unit Test

```
import Html
import List
```

```

import Set

type alias Edge comparable = (comparable, comparable)
type alias AdjList comparable = (List((comparable, List comparable)))
type alias Graph comparable = (List comparable, List (Edge comparable))

adjListToGraph : AdjList comparable -> Graph comparable
adjListToGraph (edges) =
    -- your implementation goes here
    ([], [])

main =
    Html.text
        (if (test) then
            "Your implementation passed all tests."
        else
            "Your implementation failed at least one test."
        )

test : Bool
test =
    List.all ((==) True)
        [ graph80 == adjListToGraph adjList80
        , ([], []) == adjListToGraph []
        ]

graph80 = ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ],
             [ ('b','c'), ('b','f'), ('c','f'), ('f','k'), ('g','h') ] )

adjList80 =
    [ ('b',['c','f'])

```

```
, ('c', ['b', 'f'])  
, ('d', [])  
, ('f', ['b', 'c', 'k'])  
, ('g', ['h'])  
, ('h', ['g'])  
, ('k', ['f'])  
]
```

## Solution

[Solution](#)

## Problem 81

Write a function that, given two nodes a graph, returns all the acyclic paths between the two nodes.

### Example

```
paths : (Graph a) a a -> List (List a)

g81 = ([ 'b', 'c', 'd', 'f', 'g', 'h', 'k'], [ ('b', 'c'), ('b', 'f'), ('c', 'f'), ('f', 'k'), ('g', 'h')])

paths g81 'k' 'b' == [ ['k', 'f', 'c', 'b'], ['k', 'f', 'b']]
```

### Unit Test

```
import Html
import List
import Set

-- Nodes of a graph must be of type comparable because we define

type alias Edge comparable = (comparable, comparable)
type alias AdjList comparable = (List((comparable, List comparable)))
type alias Graph comparable = (List comparable, List (Edge comparable))

-- Given two nodes a graph, return all acyclic paths in the graph between the two nodes
findPaths : comparable -> comparable -> Graph comparable -> List (List comparable)
findPaths start goal graph =
    -- your implementation goes here
```



```

[]

main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."
      1 ->
        "Your implementation failed one test."
      x ->
        "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
  List.length <| List.filter ((==) False)
    [ List.sort (findPaths 'c' 'b' graph80) == [['c','b'], ['c'
, 'f', 'b']]
      , List.sort (findPaths 'k' 'c' graph80) == [['k', 'f', 'b',
'c'], ['k', 'f', 'c']]
      , List.sort (findPaths 'x' 'y' graph80) == []
      , List.sort (findPaths 'c' 'y' graph80) == []
      , List.sort (findPaths 'y' 'c' graph80) == []
      , List.sort (findPaths 'f' 'h' graph80b) == [ ['f', 'b', '
g', 'h']
  , ['f', 'c', '
b', 'g', 'h']
  , ['f', 'g', '
h']
  ]
    ]

graph80 = ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ],
            [ ('b','c'), ('b','f'), ('c','f'), ('f','k'), ('g','
h') ] )

graph80b = ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ],

```

```
[ ('b','c'), ('b','f'), ('b','g'), ('c','f'), ('f',  
'g'), ('f','k'), ('g','h') ] )
```

## Solution

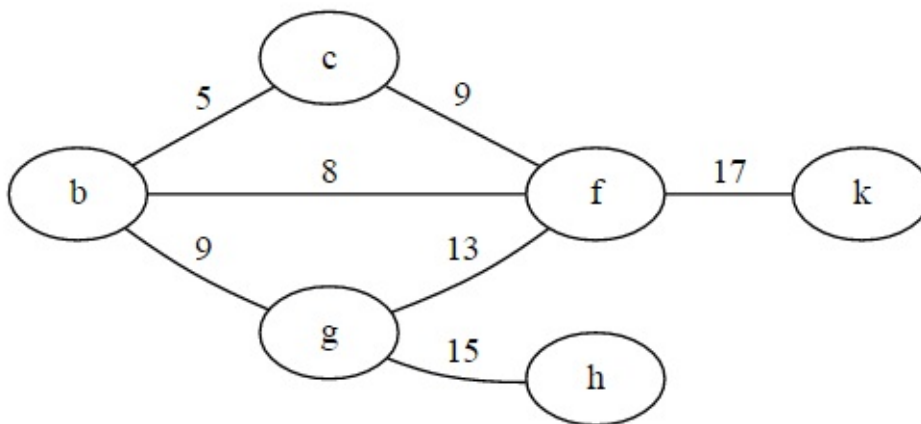
[Solution](#)

## Problem 84

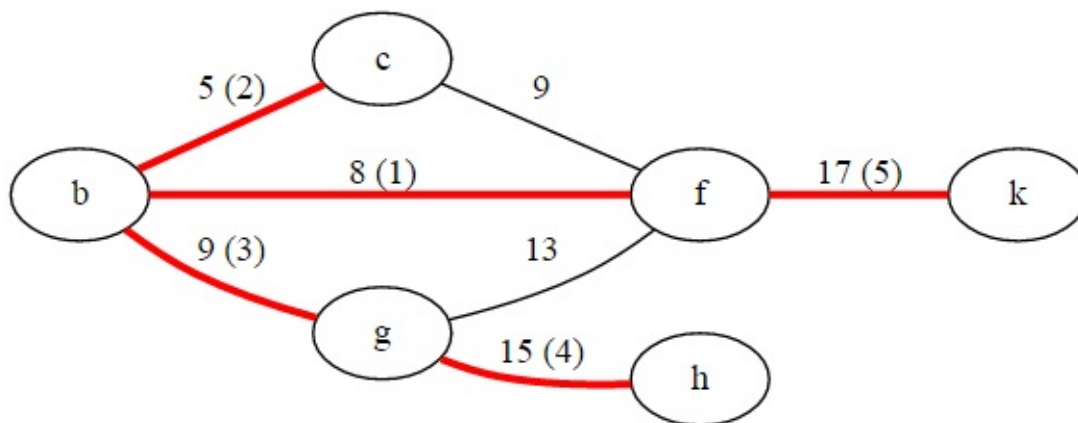
Use [Prim's algorithm](#) to find the [minimal spanning tree](#) of a [connected, edge-weighted](#) graph.

### Example

Example graph with weighted edges.



The minimum spanning tree, starting from f, and in the order of edges found by Prim's algorithm.



```
graph84 = ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ],
            [ ('b','c', 5), ('b','f', 8), ('b', 'g', 9), ('c', 'f', 9), ('f', 'g', 13), ('f','k', 17), ('g','h', 15) ] )

prim 'f' graph84 == ([ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ], [( 'b','f',8),('b','c',5),('b','g',9),('g','h',15),('f','k',17)])
```

## Unit Test

```
import Html
import List
import Set

type alias Edge comparable =
  ( comparable, comparable, Int )

type alias AdjList comparable =
  List ( comparable, List comparable )

type alias Graph comparable =
  ( List comparable, List (Edge comparable) )

type alias SpanningTree comparable =
  ( Graph comparable, Graph comparable )

{- Given a starting node and a graph with weighted edges,
   return the minimum spanning tree in a graph
-}
prim : comparable -> Graph comparable -> Graph comparable
prim start ( nodes, edges ) =
  -- your implementation goes here
  ([], [])
```

```
main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ prim 'f' graph84 == prim84
      , prim 'f' loopsAndParallels == prim84
      ]

graph84 =
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
  , [ ( 'b', 'c', 5 )
    , ( 'b', 'f', 8 )
    , ( 'b', 'g', 9 )
    , ( 'c', 'f', 9 )
    , ( 'f', 'g', 13 )
    , ( 'f', 'k', 17 )
    , ( 'g', 'h', 15 )
    ]
  )

loopsAndParallels =
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
  , [ ( 'b', 'c', 5 )
    , ( 'b', 'f', 8 )
```

```
, ( 'b', 'g', 9 )
, ( 'c', 'f', 9 )
, ( 'f', 'g', 13 )
, ( 'f', 'k', 17 )
, ( 'g', 'h', 15 )
, ( 'c', 'c', 1000)
, ( 'c', 'f', 1000)
]
)
```

```
prim84 = ( [ 'b', 'c', 'f', 'g', 'h', 'k' ]
, [ ( 'b', 'f', 8 )
, ( 'b', 'c', 5 )
, ( 'b', 'g', 9 )
, ( 'g', 'h', 15 )
, ( 'f', 'k', 17 )
]
)
```

## Solution

[Solution](#)

## Problem 86a

Set the degree of a every node in a graph. The degree of a node is the number of edges touching that node. A loop, an edge where both ends are the same node, counts as two.

We will use the degree to color a graph with the minimum number of colors. We now define a node type to have a value, degree and color.

## Example

```
{-| A node has a value, degree and a color
-}
```

```
type alias Node comparable =
  ( comparable, Int, Int )
```

```
g80 =
  ( [ ( 'b', 0, 0 )
    , ( 'c', 0, 0 )
    , ( 'd', 0, 0 )
    , ( 'f', 0, 0 )
    , ( 'g', 0, 0 )
    , ( 'h', 0, 0 )
    , ( 'k', 0, 0 )
    ]
  , [ ( 'b', 'c', 5 )
    , ( 'b', 'f', 8 )
    , ( 'c', 'f', 9 )
    , ( 'f', 'k', 17 )
    , ( 'g', 'h', 15 )
    ]
  )
```

```
g80d =
  ( [ ( 'b', 2, 0 )
    , ( 'c', 2, 0 )
```

```

    , ( 'd', 0, 0 )
    , ( 'f', 3, 0 )
    , ( 'g', 1, 0 )
    , ( 'h', 1, 0 )
    , ( 'k', 1, 0 )
  ]
, [ ( 'b', 'c', 5 )
    , ( 'b', 'f', 8 )
    , ( 'c', 'f', 9 )
    , ( 'f', 'k', 17 )
    , ( 'g', 'h', 15 )
  ]
)

```

```
degree g80 = g80d
```

## Unit Test

```

import Html
import List
import Set

type alias Node comparable =
  ( comparable, Int, Int )

type alias Edge comparable =
  ( comparable, comparable, Int )

type alias Graph comparable =
  ( List (Node comparable), List (Edge comparable) )

{-| Given a graph return a graph with the degree of each node co
rrectly set
    so we can color the graph with the minimum number of colors

```



```

-}
degree : Graph comparable -> Graph comparable
degree ( nodes, edges ) =
    -- your implementation goes here
    ([],[])

main : Html.Html a
main =
    Html.text
        <| case test of
            0 ->
                "Your implementation passed all tests."

            1 ->
                "Your implementation failed one test."

            x ->
                "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
    List.length
        <| List.filter ((==) False)
            [ degree graph84 == graph84
            , degree graph80 == graph80
            , degree graph81 == graph81
            , degree ( [], [] ) == ( [], [] )
            , degree loopsAndParallels == loopsAndParallels
            ]

g80 =
    ( [ ( 'b', 2, 2 )
      , ( 'c', 2, 2 )
      , ( 'd', 0, 0 )
      , ( 'f', 3, 3 )
      , ( 'g', 1, 1 )
    ]

```

```

    , ( 'h', 1, 1 )
    , ( 'k', 1, 1 )
  ]
, [ ( 'b', 'c', 5 )
    , ( 'b', 'f', 8 )
    , ( 'c', 'f', 9 )
    , ( 'f', 'k', 17 )
    , ( 'g', 'h', 15 )
  ]
)

```

graph80 =

```

( [ ( 'b', 2, 0 )
    , ( 'c', 2, 0 )
    , ( 'd', 0, 0 )
    , ( 'f', 3, 0 )
    , ( 'g', 1, 0 )
    , ( 'h', 1, 0 )
    , ( 'k', 1, 0 )
  ]
, [ ( 'b', 'c', 5 )
    , ( 'b', 'f', 8 )
    , ( 'c', 'f', 9 )
    , ( 'f', 'k', 17 )
    , ( 'g', 'h', 15 )
  ]
)

```

graph81 =

```

-- has a loop and parallel edges
( [ ( 'b', 2, 0 )
    , ( 'c', 2, 0 )
    , ( 'd', 0, 0 )
    , ( 'f', 3, 0 )
    , ( 'g', 4, 0 )
    , ( 'h', 2, 0 )
    , ( 'k', 1, 0 )
  ]
)

```

```
, [ ( 'b', 'c', 5 )  
    , ( 'b', 'f', 8 )  
    , ( 'c', 'f', 9 )  
    , ( 'f', 'k', 17 )  
    , ( 'g', 'h', 15 )  
    , ( 'g', 'h', 14 )  
    , ( 'g', 'g', 15 )  
  ]  
)
```

graph84 =

```
( [ ( 'b', 3, 0 )  
    , ( 'c', 2, 0 )  
    , ( 'd', 0, 0 )  
    , ( 'f', 4, 0 )  
    , ( 'g', 3, 0 )  
    , ( 'h', 1, 0 )  
    , ( 'k', 1, 0 )  
  ]  
  , [ ( 'b', 'c', 5 )  
      , ( 'b', 'f', 8 )  
      , ( 'b', 'g', 9 )  
      , ( 'c', 'f', 9 )  
      , ( 'f', 'g', 13 )  
      , ( 'f', 'k', 17 )  
      , ( 'g', 'h', 15 )  
    ]  
)
```

loopsAndParallels =

```
( [ ( 'b', 3, 0 )  
    , ( 'c', 2, 0 )  
    , ( 'd', 0, 0 )  
    , ( 'f', 4, 0 )  
    , ( 'g', 4, 0 )  
    , ( 'h', 2, 0 )  
    , ( 'k', 3, 0 )  
  ]
```

```
, [ ( 'b', 'c', 5 )  
    , ( 'b', 'f', 8 )  
    , ( 'b', 'g', 9 )  
    , ( 'c', 'f', 9 )  
    , ( 'f', 'g', 13 )  
    , ( 'f', 'k', 17 )  
    , ( 'g', 'h', 15 )  
    , ( 'h', 'g', 15 )  
    , ( 'k', 'k', 15 )  
  ]  
)
```

## Solution

[Solution](#)

## Problem 86b

Use the [Welsh-Powell algorithm](#) to color a graph with the minimum number of colors.

## Unit Test

```
import Html
import List
import Set

{-| A node has a value, degree and a color
-}
type alias Node comparable =
  ( comparable, Int, Int )

type alias Edge comparable =
  ( comparable, comparable, Int )

type alias Graph comparable =
  ( List (Node comparable), List (Edge comparable) )

{-| Given a graph return a graph with the node colors set
    with the minimum number of colors (using the Welsh-Powell al
    gorithm)
-}
colorize : Graph comparable -> Graph comparable
colorize ( nodes, edges ) =
  -- your implemenation goes here
  ( nodes, edges )
```

```

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ testColor graph84 (colorize graph84)
      , testColor graph80 (colorize graph80)
      , testColor ( [], [] ) (colorize ( [], [] ))
      ]

{-| Given a graph, return true if it properly colored.
    * The graph must have no more than 4 colors,
    * All neighboring nodes must have different colors
-}
testColor : Graph comparable -> Graph comparable -> Bool
testColor ( n1, e1 ) ( nodes, edges ) =
  let
    lessThan5 =
      List.all (\( v, d, c ) -> (c > 0) && (c < 5)) nodes

    neighborsDiffer =
      List.all (\( ca, cb ) -> ca /= cb)
        <| List.map (\( a, b, w ) -> ( (colorOfNode a nodes), (colorOfNode b nodes) )) edges

```

```

    eTest = List.map \(a,b,w) -> (a,b))
    nTest = List.map \(v,d,c) -> v)
in
    List.all ((==) True)
        [ lessThan5
          , neighborsDiffer
          , (eTest e1) == (eTest edges)
          , List.sort (nTest n1) == List.sort (nTest nodes)
        ]

colorOfNode : comparable -> List (Node comparable) -> Int
colorOfNode v nodes =
    case nodes of
        [] ->
            0

        ( vv, d, c ) :: ns ->
            if v == vv then
                c
            else
                colorOfNode v ns

graph80 =
    ( [ ( 'b', 2, 0 )
      , ( 'c', 2, 0 )
      , ( 'd', 0, 0 )
      , ( 'f', 3, 0 )
      , ( 'g', 1, 0 )
      , ( 'h', 1, 0 )
      , ( 'k', 1, 0 )
      ]
    , [ ( 'b', 'c', 5 )
      , ( 'b', 'f', 8 )
      , ( 'c', 'f', 9 )
      , ( 'f', 'k', 17 )
      , ( 'g', 'h', 15 )
      ]
    )

```

```
graph81 =  
    -- has a loop and parallel edges  
    ( [ ( 'b', 2, 0 )  
        , ( 'c', 2, 0 )  
        , ( 'd', 0, 0 )  
        , ( 'f', 3, 0 )  
        , ( 'g', 4, 0 )  
        , ( 'h', 2, 0 )  
        , ( 'k', 1, 0 )  
      ]  
    , [ ( 'b', 'c', 5 )  
        , ( 'b', 'f', 8 )  
        , ( 'c', 'f', 9 )  
        , ( 'f', 'k', 17 )  
        , ( 'g', 'h', 15 )  
        , ( 'g', 'h', 14 )  
        , ( 'g', 'g', 15 )  
      ]  
    )
```

```
graph84 =  
    ( [ ( 'b', 3, 0 )  
        , ( 'c', 2, 0 )  
        , ( 'd', 0, 0 )  
        , ( 'f', 4, 0 )  
        , ( 'g', 3, 0 )  
        , ( 'h', 1, 0 )  
        , ( 'k', 1, 0 )  
      ]  
    , [ ( 'b', 'c', 5 )  
        , ( 'b', 'f', 8 )  
        , ( 'b', 'g', 9 )  
        , ( 'c', 'f', 9 )  
        , ( 'f', 'g', 13 )  
        , ( 'f', 'k', 17 )  
        , ( 'g', 'h', 15 )  
      ]  
    )
```





## Solution

[Solution](#)


## Problem 87a

Extract nodes from a graph into a list in [depth-first order](#), without repeating any nodes. Search the children of each node in order from lowest to highest.

### Example

```
g87 = ([1,2,3,4,5,6,7], [(1,2),(2,3),(1,4),(3,4),(5,2),(5,4),(6,7)
])

depthFirst g87 == [1, 2, 3, 4, 5]
```



### Unit Test

```
import Html
import List
import Set
import String

type alias Edge comparable =
  ( comparable, comparable )

type alias Graph comparable =
  ( List comparable, List (Edge comparable) )

{-| Given a graph and a starting node, return all nodes in depth
-first order.
-}
depthFirst : comparable -> Graph comparable -> List comparable
depthFirst start (nodes, edges) =
```

```

-- your implementation goes here
[]

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ String.fromList (depthFirst 'g' graph80) == "gh"
      , String.fromList (depthFirst 'h' graph80) == "hg"
      , String.fromList (depthFirst 'g' graph84) == "gbcfk"
h"
      , String.fromList (depthFirst 'c' graph84) == "cbfgh"
k"
      , String.fromList (depthFirst 'f' graph84) == "fbcgh"
k"
      , String.fromList (depthFirst 'h' graph84) == "hgbcf"
k"
      , String.fromList (depthFirst 'k' graph84) == "kfbcg"
h"
      , String.fromList (depthFirst 'b' graph80) == "bcfk"
      , String.fromList (depthFirst 'c' graph80) == "cbfk"
      , String.fromList (depthFirst 'f' graph80) == "fbck"
      , String.fromList (depthFirst 'k' graph80) == "kfbk"
      , String.fromList (depthFirst 'k' graph80) == "kfbk"

```

```

        , String.fromList (depthFirst 'g' graph81) == "gh"
        , String.fromList (depthFirst 'h' graph81) == "hg"
        , String.fromList (depthFirst 'b' graph84) == "bcfgh"
k"
        , String.fromList (depthFirst 'a' graph87) == "abdc"
]

```

```

graph80 =
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
  , [ ( 'b', 'c' )
      , ( 'b', 'f' )
      , ( 'c', 'f' )
      , ( 'f', 'k' )
      , ( 'g', 'h' )
    ]
  )

```

```

graph81 =
  -- has a loop and parallel edges
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
  , [ ( 'b', 'c' )
      , ( 'b', 'f' )
      , ( 'c', 'f' )
      , ( 'f', 'k' )
      , ( 'g', 'h' )
      , ( 'g', 'h' )
      , ( 'g', 'g' )
    ]
  )

```

```

graph84 =
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
  , [ ( 'b', 'c' )
      , ( 'b', 'f' )
      , ( 'b', 'g' )
      , ( 'c', 'f' )
      , ( 'f', 'g' )
    ]
  )

```

```
    , ( 'f', 'k' )  
    , ( 'g', 'h' )  
  ]  
)
```

```
{-| Simple graph, in dfs - "abdc", in bfs "abcd" -}  
graph87 =
```

```
  ( [ 'a', 'b', 'c', 'd' ]  
    , [ ( 'a', 'b' )  
        , ( 'a', 'c' )  
        , ( 'b', 'd' )  
      ]  
    )
```

## Hint

1. Track of all nodes you have already visited as you traverse the graph to avoid cyclic paths.

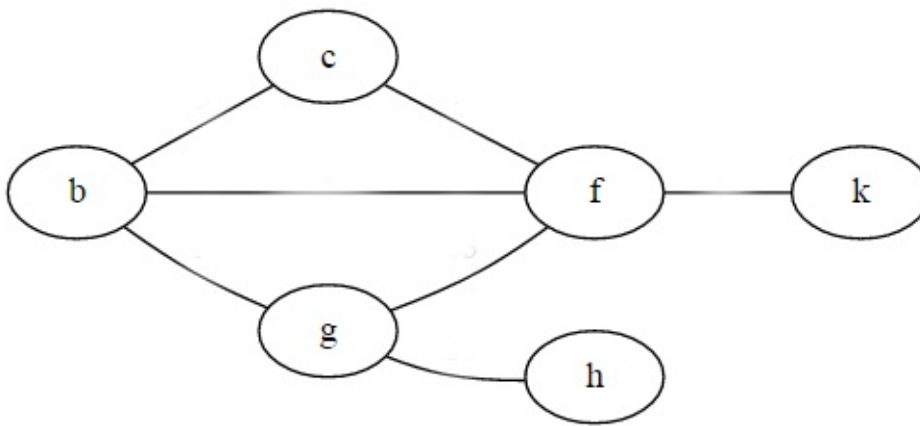
## Solution

[Solution](#)

## Problem 87b

Extract nodes from a graph into a list in [breadth-first order](#), without repeating any nodes.

### Example



```
breadthFirst 'c' graph == ['c', 'b', 'f', 'g', 'k', 'h']
```

### Unit Test

```
import Html
import List
import Set
import String

{-| A node has a value, degree and a color
-}
type alias Edge comparable =
    ( comparable, comparable )
```

```

type alias Graph comparable =
  ( List comparable, List (Edge comparable) )

{-| Given a graph and a starting node, return all nodes in breadth-first order.
-}
breadthFirst : comparable -> Graph comparable -> List comparable
breadthFirst start ( nodes, edges ) =
  -- your implementation goes here
  []

main : Html.Html a
main =
  Html.text
    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ String.fromList (breadthFirst 'a' graph87) == "abc"
        , String.fromList (breadthFirst 'g' graph80) == "gh"
        , String.fromList (breadthFirst 'h' graph80) == "hg"
        , String.fromList (breadthFirst 'g' graph84) == "gbcfkh"
        , String.fromList (breadthFirst 'c' graph84) == "cbfghk"
      ]

```

```

    , String.fromList (breadthFirst 'f' graph84) == "fbc
ghk"
    , String.fromList (breadthFirst 'h' graph84) == "hgb
cfk"
    , String.fromList (breadthFirst 'k' graph84) == "kfb
cgh"
    , String.fromList (breadthFirst 'b' graph80) == "bcf
k"
    , String.fromList (breadthFirst 'c' graph80) == "cbf
k"
    , String.fromList (breadthFirst 'f' graph80) == "fbc
k"
    , String.fromList (breadthFirst 'k' graph80) == "kfb
c"
    , String.fromList (breadthFirst 'k' graph80) == "kfb
c"
    , String.fromList (breadthFirst 'g' graph81) == "gh"
    , String.fromList (breadthFirst 'h' graph81) == "hg"
    -- either ordering is acceptable
    , List.member (String.fromList (breadthFirst 'b' gra
ph84)) [ "bcfgkh", "bcfghk" ]
    ]

```

```

graph80 =
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
  , [ ( 'b', 'c' )
    , ( 'b', 'f' )
    , ( 'c', 'f' )
    , ( 'f', 'k' )
    , ( 'g', 'h' )
    ]
  )

```

```

graph81 =
  -- has a loop and parallel edges
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
  , [ ( 'b', 'c' )
    , ( 'b', 'f' )

```



```

    , ( 'c', 'f' )
    , ( 'f', 'k' )
    , ( 'g', 'h' )
    , ( 'g', 'h' )
    , ( 'g', 'g' )
  ]
)

```

```

graph84 =
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
    , [ ( 'b', 'c' )
        , ( 'b', 'f' )
        , ( 'b', 'g' )
        , ( 'c', 'f' )
        , ( 'f', 'g' )
        , ( 'f', 'k' )
        , ( 'g', 'h' )
      ]
  )

```

```

{-| Simple graph, in dfs - "abdc", in bfs "abcd"
-}

```

```

graph87 =
  ( [ 'a', 'b', 'c', 'd' ]
    , [ ( 'a', 'b' )
        , ( 'a', 'c' )
        , ( 'b', 'd' )
      ]
  )

```

## Hint

1. A small modification to the [depth-first graph](#) search can resolve solve this problem.

## Solution

Solution

## Problem 88

Split a graph into its connected components.

### Example

```
connected ([1,2,3,4,5,6,7], [(1,2),(2,3),(1,4),(3,4),(5,2),(5,4)
, (6,7)])
== [[1,2,3,4,5][6,7]]
```

### Unit Test

```
import Html
import List
import Set
import String

type alias Edge comparable =
  ( comparable, comparable )

type alias Graph comparable =
  ( List comparable, List (Edge comparable) )

{-| Given a graph, return connected subsets of nodes.
-}
connected : Graph comparable -> List (List comparable)
connected (nodes, edges) =
  []

main : Html.Html a
main =
  Html.text
```

```

    <| case test of
      0 ->
        "Your implementation passed all tests."

      1 ->
        "Your implementation failed one test."

      x ->
        "Your implementation failed " ++ (toString x) ++
" tests."

test : Int
test =
  List.length
    <| List.filter ((==) False)
      [ (List.sort (connected graph80))
        == [ ['b', 'c', 'f', 'k']
            , ['d']
            , ['g', 'h']
            ]
        , (connected ([], [])) == []
        , (connected ([1, 2, 3], [])) == [[1], [2], [3]]
        , (connected ([1, 2, 3], [(1,2), (2,3)])) == [[1,2,3]
]]

graph80 =
  ( [ 'b', 'c', 'd', 'f', 'g', 'h', 'k' ]
    , [ ( 'b', 'c' )
        , ( 'b', 'f' )
        , ( 'c', 'f' )
        , ( 'f', 'k' )
        , ( 'g', 'h' )
        ]
    )

```

## Hint

1. Which previous graph problem will extract one connected component?

## Solution

Solution

## Problem 90

The *Eight Queens Problem* 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; that is no two queens are in the same row, the same column, or on the same diagonal.

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.

```
import Html exposing (text)

queens : List (List Int)
queens =
  -- your implementation here

main = text <| toString <| List.length queens
```

### Result

```
92
```

## Problem 91

Another famous problem is *The Knight's Tour*. The challenge is to find a path for a knight jump on an NxN chessboard such that it visits every square exactly once.

Represent the squares by pairs of integer coordinates between 1 and N and the solution as a list of positions.

```
import Html exposing (text)

knightsTour : Int -> List (Int, Int)
knightsTour n =
  -- your implementation here

main = text <| toString <| knightsTour 8
```

Possible result:

```
[(2,7),(3,5),(5,6),(4,8),(3,6),(4,4),(6,5),(4,6),
(5,4),(7,5),(6,3),(5,5),(4,3),(2,4),(1,6),(2,8),
(4,7),(6,8),(8,7),(6,6),(4,5),(6,4),(5,2),(7,1),
(8,3),(6,2),(8,1),(7,3),(8,5),(7,7),(5,8),(3,7),
(1,8),(2,6),(3,4),(1,5),(2,3),(3,1),(1,2),(3,3),
(1,4),(2,2),(4,1),(5,3),(7,4),(8,2),(6,1),(4,2),
(2,1),(1,3),(2,5),(1,7),(3,8),(5,7),(7,8),(8,6),
(6,7),(8,8),(7,6),(8,4),(7,2),(5,1),(3,2),(1,1)]
```

## Solution

A solution is [offered on Rosetta Code](#) and can be [viewed here](#).

## Problem 92

In graph theory, a graceful labeling of a graph with  $N$  edges is a labeling of its vertices with some subset of the integers between 0 and  $N$ , such that no two vertices share a label, and each edge is uniquely identified by the absolute difference between its endpoints.



A unproven conjecture in graph theory is the *Graceful Tree conjecture* or *Ringel–Kotzig conjecture*, which hypothesizes that all trees are graceful. Find a graceful labeling of this larger tree.





```
import Html exposing (text)
import Maybe

type Graph a = (List a, Edges a)

g92 = Graph ([1..14], [(1,6),(2,6),(3,6),(4,6),(5,6),(5,7),(5,8)
, (8,9),(5,10),(10,11),(11,12),(11,13),(13,14)])

graceful : Graph a -> Maybe (Graph Int)
graceful g =
  -- your implementation here

main =
  case graceful g92 of
    Just a ->
      text (toString a)

    Nothing ->
      text "Alert the media! You have found an exception
of the Graceful Tree conjecture."
```

Result:

```
[6,7,8,9,3,4,10,11,5,12,2,13,14,1]
```

## Problem 93

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.

Division should be interpreted as operating on rationals, and division by zero should be avoided.

```
[2, 3, 5, 7, 11]
```

$$2 = 3 - (5 + 7 - 11)$$

$$2 = 3 - 5 - (7 - 11)$$

$$2 = 3 - (5 + 7) + 11$$

$$2 = 3 - 5 - 7 + 11$$

$$2 = (3 * 5 + 7) / 11$$

$$2 * (3 - 5) = 7 - 11$$

$$2 - (3 - (5 + 7)) = 11$$

$$2 - (3 - 5 - 7) = 11$$

$$2 - (3 - 5) + 7 = 11$$

$$2 - 3 + 5 + 7 = 11$$

## Problem 94

In a regular graph every node has the same number of edges connecting to it. How many regular non-isomorphic graphs are there that have 6 nodes with 3 edges each?

## Problem 95

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

## Problem 96

The syntax diagram below defines the a valid form of an identifier in the Ada programming language.



`this-is-a-long-identifier` is a valid identifier. `this-ends-in-`, `two--`  
`hyphens` and `1-digit-at-start` are not.

Write a function to check if a string is a valid Ada identifier.

## Problem 97

In a sudoku puzzle, every spot in the puzzle belongs to a row and a column, as well as to one single 3x3 square. At the beginning, some of the spots already carry a number between 1 and 9. The challenge 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.

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

Write a program to solve a sudoku.

## Hint

1. You can model Sudoku puzzle as a graph with 81 nodes, one for each cell. Each node has an edge to the other nodes in its row, its column and its square.

2. What graph algorithm can you use to solve the Sudoku?
3. [A solution described](#) by [Mustafa Eissa](#).

## Problem 98

In a Nonogram 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. Around 1994, a certain kind of puzzle 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 a Prolog programmer, you are in a better situation: you can have your computer do the work! Just write a little program ;-).

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:

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

Solution:

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

Solve the nonogram.



nonogram [[3],[2,1],[3,2],[2,2],[6],[1,5],[6],[1],[2]] [[1,2],[3,1],[1,5],[7,1],[5],[3],[4],[3]]

Result

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

# Problem 99

# All Solutions

All problem solutions presented in numerical order.

# Problem 1 Solutions

## Solution 1

Recursive search for the last element

```
last : List a -> Maybe a
last xs =
  case xs of
    [ ] -> Nothing
    [a] -> Just a
    y::ys -> last ys
```

## Solution 2

Reverse and take the head.

```
last : List a -> Maybe a
last xs =
  List.reverse xs |> List.head
```

## Solution 3

[Point-free style](#), reverse and take the head.

```
last : List a -> Maybe a
last =
  List.reverse >> List.head
```

## Solution 4

Use `List.foldl1` .

```
last : List a -> Maybe a
last list =
  case list of
    [] ->
      Nothing

    x::xs ->
      Just (List.foldl (\a b -> a) x list)
```

[Back to Problem 1](#)

## Problem 2 Solutions

### Solution 1:

Recursive search for the last element

```
penultimate : List a -> Maybe a
penultimate list =
  case list of
    [ ] -> Nothing
    [y] -> Nothing
    [y, z] -> Just y
    y::ys -> penultimate ys
```

### Solution 2:

Reverse the list and take the head of the tail.

```
penultimate : List a -> Maybe a
penultimate list =
  case List.reverse list of
    [ ] -> Nothing
    y::ys -> List.head ys
```

### Solution 3:

Reverse the list, drop one and take the head.

```
penultimate : List a -> Maybe a
penultimate list =
  case drop 1 (List.reverse list) of
    [ ] -> Nothing
    y::ys -> Just y
```

[Back to Problem 2](#)

## Problem 3 Solutions

### Solution 1

```
elementAt : List a -> Int -> Maybe a
elementAt list n =
  case List.drop (n - 1) list of
    [] ->
      Nothing

    y :: ys ->
      Just y
```

[Back to Problem 2](#)



## Problem 4 Solutions

### Solution #1:

Recursive solution

```
countElements : List a -> Int
countElements list =
  case list of
    [ ] -> 0
    _ :: ys -> 1 + countElements ys
```

### Solution #2:

Recursive solution with accumulator

```
countElements : List a -> Int
countElements list =
  let
    countElements_acc list_ acc =
      case list_ of
        [ ] -> acc
        (_::xs) -> countElements_acc xs (acc + 1)
  in
    countElements_acc list 0
```

### Solution #3:

Use foldl. This is how Elm core defines `List.length` .

```
countElements : List a -> Int
countElements list = List.foldl (\_ n -> n + 1) 0 list
```

## Solution #4:

Convert elements to 1 using `List.map` and `List.sum` .

```
countElements : List a -> Int
countElements list = List.sum <| List.map (\n -> 1) list
```

[Back to Problem 4](#)

## Problem 5 Solutions

### Solution 1 - fold

```
myReverse : List a -> List a
myReverse xs =
    List.foldl (::) [] xs
```

### Solution 2 - recursion

```
myReverse : List a -> List a
myReverse list =
    case list of
        [] ->
            []

        x :: xs ->
            (myReverse xs) ++ [x]
```

[Back to Problem 5](#)

## Problem 6 Solutions

### Solution #1

```
isPalindrome : List a -> Bool
isPalindrome xs =
    List.reverse xs == xs
```

### Solution #2:

This makes half as many compares.

```
isPalindrome : List a -> Bool
isPalindrome xs =
    let
        s = (List.length xs) // 2
    in
        List.take s xs == List.take s (List.reverse xs)
```

[Back to Problem 6](#)

# Problem 7 Solution

## Solution

```
flatten : NestedList a -> List a
flatten nl =
    case nl of
        Elem x ->
            [ x ]

        SubList sl ->
            List.concatMap flatten sl
```

[Back to Problem](#)

## Extra 1 - dropWhile

```
dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
  case list of
    []      -> []

    x::xs   ->
      if (predicate x) then
        dropWhile predicate xs
      else
        list
```

[Back to problem](#)

## Extra 2 Solution - takeWhile

```
takeWhile : (a -> Bool) -> (List a) -> (List a)
takeWhile predicate xs =
  case xs of
    [] ->
      []

    hd::tl ->
      if (predicate hd) then
        hd :: takeWhile predicate tl
      else
        []
```

[Back to problem](#)

## Problem 8 Solutions

### Solution #1

```
noDupses : List a -> List a
noDupses xs =
    List.foldr noDupCons [] xs

noDupCons : a -> List a -> List a
noDupCons x xs =
    case List.head xs of
        Nothing -> [x]
        Just a   ->
            if x == a then
                xs
            else
                x :: xs
```

### Solution #2:

Define a `dropWhile` function to remove duplicates from the head of a list, then apply it recursively.



```
noDups : List a -> List a
noDups list =
  case list of
    []      -> []
    x::xs   -> x :: (noDups <| dropWhile (\u -> u == x) xs)

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
  case list of
    []      -> []

    x::xs   ->
      if (predicate x) then
        dropWhile predicate xs
      else
        list
```

[Back to Problem 8](#)

# Problem 9 Solutions

## Solution #1

A recursive solution

```
pack : List a -> List (List a)
pack xs =
  case xs of
    [] ->
      []

    y :: ys ->
      pack' y [y] ys

pack' : a -> List a -> List a -> List (List a)
pack' v run xs =
  case xs of
    [] ->
      [run]

    y :: ys ->
      if y == v then
        pack' y (y::run) ys
      else
        (run) :: pack' y [y] ys
```

## Solution #2

Using [takeWhile](#) and [dropWhile](#).

```
pack : List a -> List (List a)
pack xs =
  case xs of
    [] ->
      []

    hd::tl ->
      let
        remainder = dropWhile (\x -> x == hd) tl
      in
        takeWhile (\x -> x == hd) xs :: (pack (remainder
))
```

```
takeWhile : (a -> Bool) -> (List a) -> (List a)
takeWhile predicate xs =
  case xs of
    [] ->
      []

    hd::tl ->
      if (predicate hd) then
        hd :: takeWhile predicate tl
      else
        []
```

```
dropWhile : (a -> Bool) -> (List a) -> (List a)
dropWhile predicate list =
  case list of
    [] ->
      []

    hd::tl ->
      if (predicate hd) then
        dropWhile predicate tl
      else
        list
```

## Solution 3

Another recursive solution.

```
pack : List a -> List (List a)
pack list =
  case list of
    [] ->
      []

    x::xs ->
      let
        y = pack xs
        hd = Maybe.withDefault [] (List.head y)
      in
        if List.member x hd then
          (x::hd)::(Maybe.withDefault [] (List.tail y))
        )
        else
          [x]::y
```

## Solution 4

Fold it!

```
pack : List a -> List (List a)
pack list =
    List.foldr fPack [] list

fPack : a -> List (List a) -> List (List a)
fPack x runs =
    case runs of
        [] -> [[x]]

        y::ys ->
            case List.head y of
                Nothing -> [[x]] -- we should NEVER get here

                Just rh ->
                    if rh == x then
                        (x :: y) :: ys
                    else
                        [x] :: runs
```

[Back to Problem 9](#)

# Problem 10 Solution

## Solution 1

1. Use `List.map` to convert from `List (Int, a)` to `List (Int, Maybe a)` .
2. Use `removeNothings` to convert to `List (Int, a)` .

```
runLengths : List (List a) -> List ( Int, a )
runLengths xss =
    List.map (\xs -> ( List.length xs, List.head xs )) xss |> re
moveNothings
```

```
removeNothings : List ( Int, Maybe a ) -> List ( Int, a )
removeNothings xs =
    case xs of
        [] ->
            []

        ( c, Nothing ) :: ys ->
            removeNothings ys

        ( c, Just y ) :: ys ->
            ( c, y ) :: removeNothings ys
```

[Back to Problem 10](#)

## Problem 11 Solutions

Adapts the [takeWhile/dropWhile](#) solution from [Problem 9](#).

```
type RleCode a
  = Run Int a
  | Single a

takeWhile : (a -> Bool) -> List a -> List a
takeWhile predicate xs =
  case xs of
    [] ->
      []

    hd :: tl ->
      if (predicate hd) then
        hd :: takeWhile predicate tl
      else
        []

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
  case list of
    [] ->
      []

    hd :: tl ->
      if (predicate hd) then
        dropWhile predicate tl
      else
        list

rleEncode : List a -> List (RleCode a)
rleEncode list =
  case list of
```

```
 [] ->
   []

hd :: t1 ->
  let
    remainder =
      dropWhile (\x -> x == hd) t1

    front =
      takeWhile (\x -> x == hd) list
  in
    case List.length front of
      0 ->
        rleEncode remainder

      1 ->
        Single hd :: rleEncode remainder

      n ->
        Run n hd :: rleEncode remainder
```

[Back to Problem](#)



# Problem 12 Solutions

## Solution 1

List.concatMap makes this easy.

```
rleDecode : List (RleCode a) -> List a
rleDecode list =
    List.concatMap f list

f : RleCode a -> List a
f code =
    case code of
        Single x ->
            [ x ]

        Run n x ->
            List.repeat n x
```

[Back to problem](#)

## Problem 14 Solutions

### Solution 1 - recurse

```
duplicate : List a -> List a
duplicate list =
  case list of
    [] ->
      []

    x :: xs ->
      x :: x :: duplicate xs
```

### Solution 2 - concatMap

```
duplicate : List a -> List a
duplicate list =
  List.concatMap (\x -> [x, x]) list
```

### Solution 3 - fold

```
duplicate : List a -> List a
duplicate list =
  List.foldl (\x y -> y ++ [x, x]) [] list
```

[Back to problem](#)

## Problem 15 Solutions

### Solution 1 - recurse

```
repeatElements : Int -> List a -> List a
repeatElements n list =
  case list of
    [] -> []

    x :: xs ->
      (List.repeat n x) ++ repeatElements n xs
```

### Solution 2 - concatMap

```
repeatElements : Int -> List a -> List a
repeatElements n list =
  List.concatMap (\x -> List.repeat n x) list
```

### Solution 3 - fold

```
repeatElements : Int -> List a -> List a
repeatElements n list =
  List.foldl1 (\x y -> y ++ (List.repeat n x)) [] list
```

[Back to problem](#)

## Problem 16 Solutions

### Solution 1

```
dropNth : List a -> Int -> List a
dropNth list n =
  case max n 0 of
    0 ->
      list

    _ ->
      case list of
        [] ->
          []

        x :: xs ->
          (List.take (n - 1) list) ++ (dropNth (List.d
rop n list) n)
```

### Solution 2

Uses List.indexedMap and List.concatMap.

```
dropNth : List a -> Int -> List a
dropNth list n =
  if n < 2 then
    list
  else
    List.indexedMap (dropNth' n) list
      |> List.concatMap identity

dropNth' : Int -> Int -> a -> List a
dropNth' n i x =
  if (i + 1) % n == 0 then
    []
  else
    [ x ]
```

[Back to problem](#)

# Problem 17 Solutions

## Solution #1

Using Elm core functions `List.take` and `List.drop` .

```
split : List a -> Int -> (List a, List a)
split list count =
    (List.take count list, List.drop count list)
```

[Back to Problem 17](#)

# Problem 18 Solutions

## Solution 1

```
sublist : Int -> Int -> List a -> List a
sublist start end list =
  let
    s_ = max (start-1) 0
    e_ = max end 0
  in
    List.drop s_ list |> List.take (e_ - s_)
```

## Solution 2

Use `Array.slice`. Note that `Array.slice` uses negative indices to count from the end of the array.

```
sublist : Int -> Int -> List a -> List a
sublist start end list =
  let
    s' = max (start-1) 0
    e' = max end 0
  in
    Array.fromList list
      |> Array.slice s' e'
      |> Array.toList
```

[Back to problem](#)

## Problem 19 Solution

```
rotate : Int -> List a -> List a
rotate rot list =
  let
    r =
      rot % (List.length list)
  in
    List.drop r list ++ List.take r list
```

[Back to problem](#)



## Problem 20 Solution

```
dropAt : Int -> List a -> List a
dropAt n list =
  case list of
    [] ->
      []

    x :: xs ->
      (List.take (n - 1) list) ++ (List.drop n list)
```

[Back to problem](#)

# Problem 21 Solutions

## Solution 1

Use split from [Problem 17](#).

```
split : List a -> Int -> ( List a, List a )
split list count =
    ( List.take count list, List.drop count list )

insertAt : Int -> a -> List a -> List a
insertAt n v xs =
    let
        ( front, back ) =
            split xs (n - 1)
    in
        front ++ (v :: back)
```

## Solution 2

Do the split internally with List.drop and List.take

```
insertAt : Int -> a -> List a -> List a
insertAt n v xs =
    List.take (n - 1) xs ++ (v :: List.drop (n - 1) xs)
```

## Solution 3

Is there anything you can't do with recursion?

```
insertAt : Int -> a -> List a -> List a
insertAt n v xs =
    if n < 2 then
        v :: xs
    else
        case xs of
            [] ->
                [ v ]

            y :: ys ->
                y :: insertAt (n - 1) v ys
```

[Back to problem](#)

# Problem 22 Solutions

## Solution 1

Yeah, you can recurse this.

```
range : Int -> Int -> List Int
range start end =
  if start == end then
    [end]
  else
    if (start < end) then
      start :: range (start + 1) end
    else
      start :: range (start - 1) end
```

## Solution 2

Use the range operator (..) and handle reverse order. This is much faster

```
range : Int -> Int -> List Int
range start end =
  if start <= end then
    [start .. end]
  else
    List.reverse [end .. start]
```

[Back to problem](#)

## Problem 23 Solution

```
randomSelect : Random.Seed -> Int -> List a -> ( List a, Random.  
Seed )
```

```
randomSelect seed n list =  
  let  
    ( l, r, s ) =  
      randSelect n ( [], list, seed )  
  in  
    ( l, s )
```

```
randSelect : Int -> ( List a, List a, Random.Seed ) -> ( List a,  
List a, Random.Seed )
```

```
randSelect n ( l, r, seed ) =  
  if n > 0 then  
    let  
      ( idx, seed_ ) =  
        Random.step (Random.int 1 (List.length r)) seed  
  
      e =  
        elementAt r idx  
  
      r_ =  
        dropAt r idx  
    in  
      case e of  
        Nothing ->  
          ( l, r, seed )  
  
        Just x ->  
          randSelect (n - 1) ( x :: l, r_, seed_ )  
  else  
    ( l, r, seed )
```

```
elementAt : List a -> Int -> Maybe a
```

```
elementAt list n =  
  case List.drop (n - 1) list of  
    [] ->  
      Nothing  
  
    y :: ys ->  
      Just y  
  
dropAt : List a -> Int -> List a  
dropAt list n =  
  case list of  
    [] ->  
      []  
  
    x :: xs ->  
      (List.take (n - 1) list) ++ (List.drop n list)
```

[Back to problem](#)

## Problem 24 Solution

```
lotto : Random.Seed -> Int -> Int -> Int -> List Int
lotto seed n low high =
    List.sort
        <| fst
        <| randomSelect seed n [low .. high]
```

[Back to problem](#)

## Problem 26 Solution

```
combinations : Int -> List a -> List (List a)
combinations n list =
    if n <= 0 then
        [ [] ]
    else
        case list of
            [] ->
                []

            x :: xs ->
                List.map ((::) x) (combinations (n - 1) xs) ++ c
ombinations n xs
```

[Back to problem](#)



## Problem 28.a Solution

Could it be any simpler?

```
sortByListLengths : List (List a) -> List (List a)
sortByListLengths xs =
    List.sortBy List.length xs
```

[Back to problem](#)

## Problem 28.b

Sort list of lists by the frequency of the list length.

### Solution 1

```
sortByLengthFrequency : List (List a) -> List (List a)
sortByLengthFrequency xs =
  xs
  |> List.map (\ys -> freq ys xs )
  |> List.sortBy Tuple.first
  |> List.map Tuple.second

freq: List a -> List (List a) -> (Int, List a)
freq xs list =
  ( list
    |> List.filter (\ys -> List.length ys == List.length xs)
    |> List.length
  , xs
  )
```

### Solution 2

This solution is similar to the first, but replaces the list of length/frequency pairs with a Dict. Lengths are the dictionary keys, and frequencies are the value.

```
sortByLengthFrequency : List (List a) -> List (List a)
sortByLengthFrequency xs =
  let
    freqs =
      lenFreq xs Dict.empty
  in
    List.sortBy (sortLenFreq freqs) xs
```

```
sortLenFreq : Dict.Dict Int Int -> List a -> Int
sortLenFreq lenFreqs xs =
    case Dict.get (List.length xs) lenFreqs of
        -- should never get Nothing
        Nothing ->
            -1

        Just freq ->
            freq

lenFreq : List (List a) -> Dict.Dict Int Int -> Dict.Dict Int Int

lenFreq xs d =
    case xs of
        [] ->
            Dict.empty

        l :: ls ->
            let
                len =
                    List.length l

                ( sameLength, otherLengths ) =
                    List.partition (\x -> List.length x == len)
xs

                freq =
                    List.length sameLength

            in
                if List.isEmpty otherLengths then
                    Dict.insert len freq d
                else
                    lenFreq otherLengths (Dict.insert len freq d
    )
```

[Back to problem](#)



## Problem 31 - Sieve of Eratosthenes

```
isPrime : Int -> Bool
isPrime n =
    if n < 2 then
        False
    else
        eratos (abs n) (List.range 2 (n // 2))

eratos : Int -> List Int -> Bool
eratos n cs =
    case cs of
        [] ->
            True

        x :: xs ->
            if n % x == 0 then
                False
            else
                eratos n (List.filter (\y -> (y % x) /= 0) xs)
```

[Back to problem](#)

## Problem 32 - Greatest common denominator

```
gcd : Int -> Int -> Int
gcd a b =
    if b == 0 then
        a
    else
        gcd b (a % b)
```

[Back to problem](#)

# Problem 33 Solutions

## Solution 1

```
coprime : Int -> Int -> Bool
coprime a b =
    gcd a b == 1
```

```
gcd : Int -> Int -> Int
gcd a b =
    if b == 0 then
        abs a
    else
        gcd b (a % b)
```

[Back to problem](#)

## Problem 34 - Euler's totient

```
totient : Int -> Int
totient n =
    List.length <| List.filter (\x -> coprime n x) (List.range 1
n)

coprime : Int -> Int -> Bool
coprime a b =
    gcd a b == 1

gcd : Int -> Int -> Int
gcd a b =
    if b == 0 then
        abs a
    else
        gcd b (a % b)
```

[Back to problem](#)



## Problem 35 - Prime factors

### Solution 1

Use the Sieve of Eratosthenes to find primes, then test each prime from lowest to highest.

```
import Html
import List
import Maybe

primeFactors : Int -> List Int
primeFactors n =
    List.reverse <| factors n (primes n) []

factors : Int -> List Int -> List Int -> List Int
factors n ps acc =
    case ps of
        [] ->
            acc

        x :: xs ->
            if n == x then
                x :: acc
            else if n % x == 0 then
                factors (n // x) ps (x :: acc)
            else
                factors n xs acc

-- find all primes up to n

primes : Int -> List Int
```

```
primes n =
  if n < 2 then
    []
  else
    eratos (List.range 2 n) []

-- sieve of Eratosthenes
-- remove all the the non-primes from a list

eratos : List Int -> List Int -> List Int
eratos candidates primes =
  case candidates of
    [] ->
      List.reverse primes

    x :: xs ->
      let
        cs =
          List.filter (\y -> (y % x) /= 0) xs
      in
        eratos cs (x :: primes)
```

## Solution 2

We can optimize the first solution by eliminating primes between  $n/2$  and  $n$ .

```
primeFactors : Int -> List Int
primeFactors n =
  if n < 2 then
    []
  else
    List.reverse <| factors n ((primes (n // 2)) ++ [ n ]) [
]
```

## Solution 3

Faster still, use `dropWhile` to sieve the prime numbers for you

```
primeFactors : Int -> List Int
primeFactors n =
    if n < 2 then
        []
    else
        let
            prime =
                Maybe.withDefault 0
                    <| List.head
                    <| dropWhile (\x -> n % x /= 0) [2..n]
        in
            prime :: (primeFactors <| n // prime)

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
    case list of
        [] ->
            []

        x :: xs ->
            if (predicate x) then
                dropWhile predicate xs
            else
                list
```

[Back to problem](#)

# Problem 36 Solutions

## Solution 1

Combine code from problems [9](#), [10](#) and [35](#).

```
primeFactorsM : Int -> List (Int, Int)
primeFactorsM n =
    toTuples <| primeFactors n

toTuples : List a -> List ( a, Int )
toTuples fs =
    case fs of
        [] ->
            []

        x :: xs ->
            ( x, List.length (takeWhile (\y -> y == x) fs) )
            :: toTuples (dropWhile (\y -> y == x) fs)

primeFactors : Int -> List Int
primeFactors n =
    if n < 2 then
        []
    else
        let
            prime =
                Maybe.withDefault 0
                <| List.head
                <| dropWhile (\x -> n % x /= 0) (List.range
2 n)
        in
            prime :: (primeFactors <| n // prime)
```

```
takeWhile : (a -> Bool) -> List a -> List a
takeWhile predicate xs =
  case xs of
    [] ->
      []

    hd :: tl ->
      if (predicate hd) then
        hd :: takeWhile predicate tl
      else
        []

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
  case list of
    [] ->
      []

    x :: xs ->
      if (predicate x) then
        dropWhile predicate xs
      else
        list
```

[Back to problem](#)

## Problem 37 - Euler's totient function optimized

```

phi : Int -> Int
phi n =
  if n < 1 then
    0
  else
    List.product
      <| List.map (\( p, m ) -> (p - 1) * p ^ (m - 1))
      <| primeFactorsM n

primeFactorsM : Int -> List ( Int, Int )
primeFactorsM n =
  toTuples <| primeFactors n

primeFactors : Int -> List Int
primeFactors n =
  if n < 2 then
    []
  else
    let
      prime =
        Maybe.withDefault 0
          <| List.head
          <| dropWhile (\x -> n % x /= 0) (List.range 2
n)
    in
      prime :: (primeFactors <| n // prime)

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
  case list of
    [] ->

```

```
    []

x :: xs ->
  if (predicate x) then
    dropWhile predicate xs
  else
    list

takeWhile : (a -> Bool) -> List a -> List a
takeWhile predicate xs =
  case xs of
    [] ->
      []

  hd :: tl ->
    if (predicate hd) then
      hd :: takeWhile predicate tl
    else
      []

toTuples : List a -> List ( a, Int )
toTuples fs =
  case fs of
    [] ->
      []

  x :: xs ->
    ( x, List.length (takeWhile (\y -> y == x) fs) )
    :: toTuples (dropWhile (\y -> y == x) fs)
```

[Back to problem](#)

# Problem 38 Solutions



## Problem 39 Solutions

```
primesInRange : Int -> Int -> List Int
primesInRange low high =
    if high < low || high < 2 then
        []
    else
        dropWhile (\x -> x < low) <| primes high

-- find all primes up to n
primes : Int -> List Int
primes n =
    if n < 2 then
        []
    else
        eratos (List.range 2 n) []

-- sieve of Eratosthenes
-- remove all the the non-primes from a list
eratos : List Int -> List Int -> List Int
eratos candidates primes =
    case candidates of
        [] -> List.reverse primes

    x::xs ->
        let
            cs = List.filter (\y -> (y % x) /= 0) xs
        in
            eratos cs (x :: primes)

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
    case list of
        [] -> []
```

```
x::xs  ->
  if (predicate x) then
    dropWhile predicate xs
  else
    list
```

[Back to problem](#)

# Problem 40 Solutions

## Solution 1

```
goldbach : Int -> Maybe (Int, Int)
goldbach n =
    if isOdd n || n < 3 then
        Nothing
    else
        goldbach' n <| primesInRange 2 n

goldbach' : Int -> List Int -> Maybe (Int, Int)
goldbach' n ps =
    case ps of
        [] ->
            Nothing

        p1 :: xs ->
            let
                ps' = dropWhile (\y -> p1 + y < n) ps
                ps'' = takeWhile (\y -> p1 + y == 0) ps'
            in
                case List.head ps'' of
                    Nothing ->
                        goldbach' n xs

                    Just p2 ->
                        if p2 + p1 == n then
                            Just (p1, p2)
                        else
                            goldbach' n xs

primesInRange : Int -> Int -> List Int
primesInRange low high =
    if high < low || high < 2 then
```

```
    []
  else
    dropWhile (\x -> x < low) <| primes high

-- find all primes up to n
primes : Int -> List Int
primes n =
  if n < 2 then
    []
  else
    eratos [2..n] []

-- sieve of Eratosthenes
-- remove all the the non-primes from a list
eratos : List Int -> List Int -> List Int
eratos candidates primes =
  case candidates of
    [] -> List.reverse primes

  x::xs ->
    let
      cs = List.filter (\y -> (y % x) /= 0) xs
    in
      eratos cs (x :: primes)

dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
  case list of
    [] -> []

  x::xs ->
    if (predicate x) then
      dropWhile predicate xs
    else
      list
```

```
takeWhile : (a -> Bool) -> List a -> List a
takeWhile predicate list =
  case list of
    []      -> []

    x::xs   ->
      if (predicate x) then
        x:: takeWhile predicate xs
      else
        list
```

[Back to problem](#)

## Problem 41 Solution

```
goldbachGT : Int -> Int -> Int -> List ( Int, Int )
goldbachGT low high limit =
    List.filter \( x, y ) -> x > limit && y > limit)
        <| List.concatMap maybeToList
        <| List.map goldbach
        <| [min low (limit * 2)..high]
```

```
maybeToList : Maybe a -> List a
maybeToList x =
    case x of
        Nothing ->
            []

        Just y ->
            [ y ]
```

```
goldbach : Int -> Maybe ( Int, Int )
goldbach n =
    if isOdd n || n < 3 then
        Nothing
    else
        goldbach' n <| primesInRange 2 n
```

```
goldbach' : Int -> List Int -> Maybe ( Int, Int )
goldbach' n ps =
    case ps of
        [] ->
            Nothing

        p1 :: xs ->
            let
                ps' =
```

```
        dropWhile (\y -> p1 + y < n) ps

    ps'' =
        takeWhile (\y -> p1 + y == 0) ps'
in
    case List.head ps'' of
        Nothing ->
            goldbach' n xs

        Just p2 ->
            if p2 + p1 == n then
                Just ( p1, p2 )
            else
                goldbach' n xs

primesInRange : Int -> Int -> List Int
primesInRange low high =
    if high < low || high < 2 then
        []
    else
        dropWhile (\x -> x < low) <| primes high

-- find all primes up to n

primes : Int -> List Int
primes n =
    if n < 2 then
        []
    else
        eratos [2..n] []

-- sieve of Eratosthenes
-- remove all the the non-primes from a list
```

```
eratos : List Int -> List Int -> List Int
eratos candidates primes =
  case candidates of
    [] ->
      List.reverse primes

    x :: xs ->
      let
        cs =
          List.filter (\y -> (y % x) /= 0) xs
      in
        eratos cs (x :: primes)
```

```
dropWhile : (a -> Bool) -> List a -> List a
dropWhile predicate list =
  case list of
    [] ->
      []

    x :: xs ->
      if (predicate x) then
        dropWhile predicate xs
      else
        list
```

```
takeWhile : (a -> Bool) -> List a -> List a
takeWhile predicate list =
  case list of
    [] ->
      []

    x :: xs ->
      if (predicate x) then
        x :: takeWhile predicate xs
      else
        list
```



```
-- The core library function Arithmetic.isOdd is not available
-- on elm-lang.org/try, so we'll recreate it here
```

```
isOdd : Int -> Bool
```

```
isOdd n =
```

```
    n % 2 /= 0
```

```
isPrime : Int -> Bool
```

```
isPrime n =
```

```
    if n < 2 then
```

```
        False
```

```
    else
```

```
        List.member n <| primesInRange 2 n
```

[Back to problem](#)

# Problem 46 Solutions

## Solution 1

```
-- True if and only if a and b are true
and' : Bool -> Bool -> Bool
and' a b =
    a && b

-- True if either a or b are true
or' : Bool -> Bool -> Bool
or' a b =
    a || b

-- True either a or b are false
nand' : Bool -> Bool -> Bool
nand' a b =
    not (a && b)

-- True if and only if a and b are false
nor' : Bool -> Bool -> Bool
nor' a b =
    not (a || b)

-- True if a or b is true, but not if both are true
xor' : Bool -> Bool -> Bool
xor' a b =
    not (a == b)

-- True if a is false or b is true
implies : Bool -> Bool -> Bool
implies a b =
```

```
    if a then b else True

-- True if both a and b are true, or both a and b are false
equivalent : Bool -> Bool -> Bool
equivalent a b =
    a == b

truthTable : (Bool -> Bool -> Bool) -> List (Bool, Bool, Bool)
truthTable f =
    -- your implementation goes here
    [ (True, True, f True True)
    , (True, False, f True False)
    , (False, True, f False True)
    , (False, False, f False False)
    ]
```

## Solution 2

It is common for functional programmers to write functions as a composition of other functions, never mentioning the actual arguments they will be applied to. This is called "[pointfree style](#)".

```
-- True if and only if a and b are true
and' : Bool -> Bool -> Bool
and' =
    (&&)

-- True if either a or b are true
or' : Bool -> Bool -> Bool
or' =
    (||)

-- True if both a and b are true, or both a and b are false
equivalent : Bool -> Bool -> Bool
equivalent =
    (==)
```

[Back to problem](#)

# Problem 47 Solutions

## Solution 1

```
-- True if and only if a and b are true
(&) : Bool -> Bool -> Bool
(&) a b =
    a && b

-- True if either a or b are true
(|) : Bool -> Bool -> Bool
(|) a b =
    a || b

-- True either a or b are false
(/&) : Bool -> Bool -> Bool
(/&) a b =
    not (a && b)

-- True if and only if a and b are false
(/|) : Bool -> Bool -> Bool
(/|) a b =
    not (a || b)

-- True if a or b is true, but not if both are true
(*|) : Bool -> Bool -> Bool
(*|) a b =
    not (a == b)

-- True if a is false or b is true
(.^) : Bool -> Bool -> Bool
(.^) a b =
```

```
    if a then b else True

-- True if both a and b are true, or both a and b are false
(==) : Bool -> Bool -> Bool
(==) a b =
    a == b

truthTable : (Bool -> Bool -> Bool) -> List (Bool, Bool, Bool)
truthTable f =
    -- your implementation goes here
    [ (True, True, f True True)
    , (True, False, f True False)
    , (False, True, f False True)
    , (False, False, f False False)
    ]
```

[Back to problem](#)

# Problem 49 Solutions

## Solution 1

```
grayCodes : Int -> List (List Int)
grayCodes numBits =
    let
        n =
            max 0 numBits
    in
        case n of
            0 ->
                []

            1 ->
                [ [ 0 ], [ 1 ] ]

            _ ->
                let
                    n =
                        abs numBits

                    xs =
                        grayCodes (n - 1)
                in
                    (List.map ((::) 0) xs) ++ (List.map ((::) 1)
(List.reverse xs))
```

[Back to problem](#)

# Problem 50a Solutions

## Solution 1

Sort the input, then use the `pack` function from [Problem 9](#), `runLengths` from [Problem 10](#). Note this requires that the type of the input be a `comparable`.

```
freqs : List comparable -> List ( Int, comparable )
freqs list =
    runLengths <| pack <| List.sort list
```

```
pack : List a -> List (List a)
pack xs =
    case xs of
        [] ->
            []

        y :: ys ->
            pack_ y [ y ] ys
```

```
pack_ : a -> List a -> List a -> List (List a)
pack_ v run xs =
    case xs of
        [] ->
            [ run ]

        y :: ys ->
            if y == v then
                pack_ y (y :: run) ys
            else
                (run) :: pack_ y [ y ] ys
```

```
runLengths : List (List a) -> List ( Int, a )
```



```
runLengths xss =  
  let  
    f : List a -> List (Int, a) -> List (Int, a)  
    f run runs =  
      case List.head run of  
        Nothing ->  
          runs  
  
        Just x ->  
          (List.length run, x) :: runs  
  in  
    List.foldl f [] xss
```

[Back to problem](#)

## Problem 54 Solution

This is the type we'll be using for binary trees in the later problems.

```
type Tree a
  = Empty
  | Node a (Tree a) (Tree a)
```

[Back to problem](#)

## Problem 55

### Solution 1

Use List.foldl, adds one node at a time.

```
balancedTree : Int -> Tree Char
balancedTree n =
    List.foldl (addBalancedNode) Empty <| List.repeat n 'x'

addBalancedNode : a -> Tree a -> Tree a
addBalancedNode v tree =
    case tree of
        Empty ->
            Node v Empty Empty

        Node v' left right ->
            if count left > count right then
                Node v' left (addBalancedNode v right)
            else
                Node v' (addBalancedNode v left) right

-- count number of Nodes in a Tree
count : Tree a -> Int
count tree =
    case tree of
        Empty -> 0

        Node n left right ->
            1 + (count left) + (count right)
```

### Solution 2

A more efficient solution adds many nodes at once.

```
import Html
import List
import String

type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

balancedTree : Int -> Tree Char
balancedTree n =
  if n < 1 then
    Empty
  else
    case n of
      1 ->
        Node 'x' Empty Empty
      n ->
        let
          n' = (n-1) // 2
        in
          if n % 2 == 1 then
            Node 'x' (balancedTree n') (balancedTree
n')
          else
            Node 'x' (balancedTree (n' + 1)) (balanc
edTree n')
```

[Back to problem](#)

## Problem 56 Solution

```
type Tree a
  = Empty
  | Node a (Tree a) (Tree a)

mirror : Tree a -> Tree a -> Bool
mirror left right =
  case left of
    Empty ->
      case right of
        Empty ->
          True
        _ ->
          False
    Node ln ll lr ->
      case right of
        Empty ->
          False
        Node rn rl rr ->
          (mirror ll rr) && (mirror lr rl)

isSymmetric : Tree a -> Bool
isSymmetric tree =
  case tree of
    Empty ->
      True

    Node n left right ->
      mirror left right
```

# Problem 57 Solutions

## Solution 1

Add one node at a time.

```
toBSTree : List comparable -> Tree comparable
toBSTree list =
    case list of
        [] ->
            Empty

        _ ->
            List.foldl addBSNode Empty list

addBSNode : comparable -> Tree comparable -> Tree comparable
addBSNode v tree =
    case tree of
        Empty ->
            Node v Empty Empty

        Node v_ left right ->
            if v > v_ then
                Node v_ left (addBSNode v right)
            else if v < v_ then
                Node v_ (addBSNode v left) right
            else
                tree
```

## Solution 2

Here's an modification using `Basics.compare` .

```
toBSTree : List comparable -> Tree comparable
toBSTree list =
  case list of
    [] ->
      Empty

    _ ->
      List.foldl addBSNode Empty list

addBSNode : comparable -> Tree comparable -> Tree comparable
addBSNode v tree =
  case tree of
    Empty ->
      Node v Empty Empty

    Node v_ left right ->
      case compare v v_ of
        GT ->
          Node v_ left (addBSNode v right)

        LT ->
          Node v_ (addBSNode v left) right

        EQ ->
          tree
```

[Back to problem](#)

# Problem 61a Solutions

## Solution 1

```
countLeaves : Tree a -> Int
countLeaves tree =
    case tree of
        Empty ->
            0

        Node v left right ->
            if left == Empty && right == Empty then
                1
            else
                (countLeaves left) + (countLeaves right)
```

[Back to problem](#)



# Problem 61b Solutions

## Solution 1

```
getLeaves : Tree a -> List a
getLeaves tree =
    case tree of
        Empty ->
            []

        Node v left right ->
            if left == Empty && right == Empty then
                [v]
            else
                getLeaves left ++ getLeaves right
```

[Back to problem](#)

# Problem 62a Solutions

## Solution 1

```
countInternals : Tree a -> Int
countInternals tree =
    case tree of
        Empty ->
            0

        Node v left right ->
            if left == Empty && right == Empty then
                0
            else
                1 + countInternals left + countInternals right
```

[Back to problem](#)

## Problem 62b Solutions

```
getInternals : Tree a -> List a
getInternals tree =
  case tree of
    Empty ->
      []

    Node v left right ->
      if left == Empty && right == Empty then
        []
      else
        getInternals left ++ v :: getInternals right
```

[Back to problem](#)

# Problem 63 Solutions

## Solution 1

```
completeTree : a -> Int -> Tree a
completeTree v n =
    case n of
        0 ->
            Empty

        _ ->
            Node v (addNode v (2) n) (addNode v (3) n)

addNode : a -> Int -> Int -> Tree a
addNode v idx end =
    if (idx == 0) || (idx > end) then
        Empty
    else
        Node v (addNode v (idx * 2) end) (addNode v ((idx * 2) +
1) end)
```

[Back to problem](#)

# Problem 64 Solutions

## Solution 1

Set the x and y coordinates in a single pass using in-order traversal.

```
layout : Tree a -> Tree (a, (Int, Int))
layout tree =
    layoutAux 1 1 tree
    |> Tuple.first

layoutAux : Int -> Int -> Tree a -> (Tree (a,(Int,Int)), Int)
layoutAux x y tree =
    case tree of
        Empty ->
            (Empty, x)

        Node n l r ->
            let
                (lTree, x1) = layoutAux x (y+1) l
                (rTree, xr) = layoutAux (x1+1) (y+1) r
            in
                (Node (n, (x1,y)) lTree rTree, xr)
```

---

## Solution 2

This solution breaks down the process into three steps,

- Inserting a coordinate to each node

```
treeMap addXY tree
```

- Setting the y value to the depth

```
treeMapDepth setY 1
```

- Setting the x value to the in-order index

```
treeMapInOrder setX 1
```

This is not an efficient solution, as the tree must be traversed multiple times. However the traversal functions `treeMapDepth` and `treeMapInOrder` modeled on `List.indexedMap` will be useful functions for other problems.

```
layout : Tree a -> Tree (a, (Int, Int))
layout tree =
  treeMap addXY tree
    |> treeMapDepth setY 1
    |> treeMapInOrder setX 1

addXY : a -> (a, (Int, Int))
addXY v =
  (v, (0, 0))

setY : Int -> (a, (Int, Int)) -> (a, (Int, Int))
setY n (v, (x, y)) =
  (v, (x, n))

setX : Int -> (a, (Int, Int)) -> (a, (Int, Int))
setX n (v, (x, y)) =
  (v, (n, y))

treeMap : (a -> b) -> Tree a -> Tree b
treeMap f tree =
  case tree of
    Empty ->
      Empty
```

```
Node v left right ->
    Node (f v) (treeMap f left) (treeMap f right)

-- apply a function to each node, passing the depth as a parameter
treeMapDepth : (Int -> a -> b) -> Int -> Tree a -> Tree b
treeMapDepth f d tree =
    case tree of
        Empty ->
            Empty

        Node v left right ->
            Node (f d v)
                (treeMapDepth f (d + 1) left)
                (treeMapDepth f (d + 1) right)

-- apply a function to each node, passing the in-order index as
a parameter
treeMapInOrder : (Int -> a -> b) -> Int -> Tree a -> Tree b
treeMapInOrder f n tree =
    case tree of
        Empty ->
            Empty

        Node v left right ->
            let
                nn = (n + (treeCount left))
            in
                Node (f nn v)
                    (treeMapInOrder f n left)
                    (treeMapInOrder f (nn + 1) right)

-- count number of Nodes in a Tree
treeCount : Tree a -> Int
treeCount tree =
    case tree of
```

```
Empty -> 0
```

```
Node n left right ->
```

```
1 + (treeCount left) + (treeCount right)
```

[Back to problem](#)



## Problem 65

```

{-| Given a Tree return a tree that adds x, y coordinates for ea
ch node
    to layout a graphic representation of the tree
-}
layout : Tree comparable -> Tree ( comparable, (Int, Int) )
layout tree =
    let
        ld = leftDepth 0 tree
        x = 2 ^ ld - 1
        sep = x
        sep_ = sep // 2
    in
        case tree of
            Empty ->
                Empty

            Node n left right ->
                Node ( n, (x, 1) )
                    (layout_ (x - sep_ - 1) 2 sep_ left)
                    (layout_ (x + sep_ + 1) 2 sep_ right)

layout_ : Int -> Int -> Int -> Tree comparable -> Tree ( compara
ble, (Int, Int) )
layout_ x y sep tree =
    let
        sep_ = sep // 2
    in
        case tree of
            Empty ->
                Empty

            Node n left right ->
                Node ( n, (x, y) )
                    (layout_ (x - sep_ - 1) (y + 1) sep_ left)

```

```
                                (layout_ (x + sep_ + 1) (y + 1) sep_ right)

depth : Int -> Tree a -> Int
depth d tree =
    case tree of
        Empty ->
            d

        Node v left right ->
            max (depth (d + 1) left) (depth (d + 1) right)

-- apply a function to each node, passing the depth as a parameter
leftDepth : Int -> Tree a -> Int
leftDepth d tree =
    case tree of
        Empty ->
            d

        Node v left right ->
            max (leftDepth (d + 1) left) (leftDepth (d + 1) right)
t)
```

[Back to problem](#)

## Problem 67

```
treeToString : Tree a -> String
treeToString tree =
  case tree of
    Empty ->
      ""

    Node v Empty Empty ->
      String.filter noQuotes (toString v)

    Node v left right ->
      String.filter noQuotes
        ((toString v)
         ++ "("
         ++ treeToString left
         ++ ","
         ++ treeToString right
         ++ ")")

noQuotes : Char -> Bool
noQuotes c =
  (c /= '\\' ) && (c /= '\"')
```

[Back to problem](#)

## Problem 68a

### Solution 1

```
preorder : Tree a -> List a
preorder tree =
  case tree of
    Empty ->
      []

    Node v l r ->
      v :: (preorder l) ++ (preorder r)
```

[Back to problem](#)

# Problem 68b Solutions

## Solutions

```
inorder : Tree a -> List a
inorder tree =
  case tree of
    Empty ->
      []

    Node v l r ->
      (inorder l) ++ (inorder r) ++ [v]
```

[Back to problem](#)

## Problem 69 Solution

```
dotToTree : String -> (Tree Char, String)
dotToTree s =
    let
        (x, xs) = Maybe.withDefault ('.', "") <| String.uncons s
    in
        case x of
            '.' ->
                (Empty, xs)

            _ ->
                let
                    (treeL, restL) = dotToTree xs
                    (treeR, restR) = dotToTree restL
                in
                    (Node x treeL treeR, restR)

treeToDot : Tree Char -> String
treeToDot tree =
    case tree of
        Empty ->
            "."

        Node v l r ->
            String.fromChar v ++ treeToDot l ++ treeToDot r
```

[Back to problem](#)

## Problem 70a

We will use this definition of a multiway tree in later problems

```
count : MTree a -> Int
count (MNode value list) =
    1 + List.sum (List.map count list)
```

[Back to problem](#)

## Problem 70b Solutions

import Html import List import Maybe import String

```
stringToTree : String -> MTree Char
stringToTree s =
  let
    (x, xs) = Maybe.withDefault ('?', "") <| String.uncons s
  in
    MNode x (Tuple.first (sToTrees xs))

sToTrees : String -> ((List (MTree Char)), String)
sToTrees s =
  let
    (x, xs) = Maybe.withDefault ('?', "") <| String.uncons s
  in
    case x of
      '^' ->
        ([], xs)

      _ ->
        let
          (list1, rest1) = sToTrees xs
          (list2, rest2) = sToTrees rest1
        in
          ([ (MNode x list1) ] ++ list2, rest2)

treeToString : MTree Char -> String
treeToString tree =
  t2s tree ++ "^"

t2s : MTree Char -> String
t2s tree =
  case tree of
    (MNode v []) ->
      String.fromChar v
```



```
(MNode v list) ->
  String.cons v
    <| (String.concat
        (List.intersperse
          "^" (List.map t2s list)))
      ++  "^"
```

[Back to problem](#)

## Problem 71 Solution

```
internalPathLength : MTree a -> Int
internalPathLength mtree =
    let
        ipl depth (MNode v list) =
            if List.length list == 0 then
                depth
            else
                depth + List.sum (List.map (ipl (depth + 1)) lis
t)
    in
        ipl 0 mtree
```

## Problem 72

Collect the nodes of a multiway tree in depth-first order.

## Solution

```
type MTree a
  = MNode a (List (MTree a))

toList : MTree a -> List a
toList (MNode v list) =
  if List.length list == 0 then
    [ v ]
  else
    (List.concatMap (\x -> toList x) list) ++ [ v ]
```

[Back to problem](#)

## Problem 80a Solution

```
graphToAdjList : Graph comparable -> AdjList comparable
graphToAdjList (nodes, edges) =
    List.map (\n -> (n, (edgesOf n edges))) nodes

edgesOf : comparable -> List (comparable, comparable) -> List comparable
edgesOf n edges =
    let
        (xs, ys) = List.unzip
        <| List.filter (\(x, y) -> x == n || y == n) edges
    in
        List.sort
            <| Set.toList
            <| Set.fromList
            <| List.filter (\x -> x /= n)
            <| xs ++ ys
```

[Back to problem](#)

## Problem 80b Solutions

```
adjListToGraph : AdjList comparable -> Graph comparable
adjListToGraph list =
    let
        nodes = List.map Tuple.first list
        edges = List.concatMap buildEdges list
                |> Set.fromList
                |> Set.toList
    in
        (nodes, edges)

buildEdges : (comparable, List comparable) -> List (Edge comparable)
buildEdges (n, list) =
    List.map (\x -> (min x n, max x n)) list
```

[Back to problem](#)

## Problem 81 Solution

This solution defines a new type, `GraphPath` which defines a path through a graph and the unused edges. Using `GraphPath` we can recurse to find all possible paths.

```
import Html
import List
import Set

-- Nodes of a graph must be of type comparable because we define

type alias Edge comparable = (comparable, comparable)
type alias AdjList comparable = (List((comparable, List comparable)))
type alias Graph comparable = (List comparable, List (Edge comparable))

-- add a path to Graph. Edges should be in either the path or the graph but never both
type alias GraphPath comparable = (Graph comparable, List comparable)

-- Given two nodes a graph, return all acyclic paths in the graph between the two nodes
findPaths : comparable -> comparable -> Graph comparable -> List (List comparable)
findPaths start goal graph =
    List.map Tuple.second <| findPaths_ goal [(graph, [start])]

-- Given a goal node and a list of path, recursively search each path
-- so we can find all complete paths to the goal
findPaths_ : comparable -> List (GraphPath comparable) -> List (GraphPath comparable)
```

```

findPaths_ goal paths =
  case paths of
    [] -> []

    p :: ps ->
      if endsAt goal p then
        p :: findPaths_ goal ps
      else
        findPaths_ goal ((extendPath p) ++ ps)

-- given a path, return a list of all edges connecting to the end of the path
extendPath : GraphPath comparable -> List (GraphPath comparable)
extendPath ((gNodes, edges), pathNodes) =
  case last pathNodes of
    Nothing ->
      []

    Just x ->
      let
        es = List.filter (\(a,b) -> (a == x) || (b == x))
          edges
        gs = List.repeat (List.length es) ((gNodes, edges), pathNodes)
      in
        List.filter isAcyclic
          <|List.map2 (\gp e -> addToPath gp e x) gs es

isAcyclic : GraphPath comparable -> Bool
isAcyclic (g, ns) =
  List.length ns == Set.size (Set.fromList ns)

--given a path, and an edge, construct a GraphPath that adds the
  edge to the path, and updates the graph to remove the edge
addToPath : GraphPath comparable -> Edge comparable -> comparable
  e -> GraphPath comparable

```

```
addToPath ((ns, es), ps) (x,y) end =  
  let  
    es_ = Set.toList <| Set.remove (x,y) <| Set.fromList es  
    end_ = if x == end then y else x  
  in  
    ((ns, es_), ps ++ [end_])  
  
-- given a value and a path, return if the value matches the las  
t node of the path  
endsAt : comparable -> GraphPath comparable -> Bool  
endsAt end (g, ns) =  
  case last ns of  
    Nothing ->  
      False  
  
    Just x ->  
      x == end  
  
last : List a -> Maybe a  
last list =  
  List.head (List.reverse list)
```

[Back to problem](#)



## Problem 84 Solution

```

prim : comparable -> Graph comparable -> Graph comparable
prim start ( nodes, edges ) =
    snd <| findMst ( ( nodes, edges ), ( [ start ], [] ) )

{-| given a SpanningTree, add the lowest weight edge from the left
graph
    to the right graph until it includes all nodes
-}
findMst : SpanningTree comparable -> SpanningTree comparable
findMst ( ( lNodes, lEdges ), ( rNodes, rEdges ) ) =
    let
        edges =
            List.filter (hasOneNodeIn rNodes) lEdges

        edge =
            minimumBy \( a, b, w ) -> w) edges
    in
        case edge of
            Nothing ->
                ( ( lNodes, lEdges ), ( rNodes, rEdges ) )

            Just e ->
                let
                    rEdges' =
                        rEdges ++ [ e ]

                    rNodes' =
                        nodesOfEdges rEdges'

                    lEdges' =
                        Set.toList <| Set.remove e <| Set.fromList
st lEdges
                in
                    if List.length lNodes <= List.length rNodes

```

```

then
    ( ( lNodes, lEdges ), ( rNodes', rEdges'
  ) )
else
    findMst ( ( lNodes, lEdges' ), ( rNodes'
, rEdges' ) )

hasOneNodeIn : List comparable -> Edge comparable -> Bool
hasOneNodeIn nodes ( a, b, w ) =
    (List.member a nodes && (not (List.member b nodes)))
    || ((not (List.member a nodes)) && List.member b nodes)

nodesOfEdges : List (Edge comparable) -> List comparable
nodesOfEdges edges =
    List.sort
    <| Set.toList
    <| Set.fromList
    <| List.concatMap (\( a, b, w ) -> [ a, b ]) edges

{-| Find the first minimum element in a list using a comparable
transformation
    (borrowed from List.Extra)
-}
minimumBy : (a -> comparable) -> List a -> Maybe a
minimumBy f ls =
    let
        minBy x ( y, fy ) =
            let
                fx =
                    f x
            in
                if fx < fy then
                    ( x, fx )
                else
                    ( y, fy )
    in

```

```
case ls of
  [ l' ] ->
    Just l'

  l' :: ls' ->
    Just <| fst <| List.foldl minBy ( l', f l' ) ls'

  _ ->
    Nothing
```

```
last : List a -> Maybe a
last list =
  List.head (List.reverse list)
```

[Back to problem](#)

## Problem 86b

```
{-| Given a graph return a graph with the node colors set
      with the minimum number of colors (using the Welsh-Powell al
      gorithm)
-}
colorize : Graph comparable -> Graph comparable
colorize ( nodes, edges ) =
    let
        defColor =
            List.length nodes

        ( ns, es ) =
            sortByDegree
                <| degree defColor ( nodes, edges )
    in
        colorize_ ( ns, es )

{-| Given a graph return a graph with the node colors set
      with the minimum number of colors (using the Welsh-Powell al
      gorithm)
-}
colorize_ : Graph comparable -> Graph comparable
colorize_ ( nodes, edges ) =
    let
        ( colored, uncolored ) =
            List.partition (\( v, d, c ) -> c < List.length node
s) nodes
    in
        case uncolored of
            [] ->
                ( colored, edges )

            ( v, d, c ) :: ns ->
                let
                    colorNode =
```

```

        ( v, d, lowColor v ( nodes, edges ) )

    nodes_ =
        colored ++ (colorNode :: ns)
    in
        colorize_ ( nodes_, edges )

{-| Given a node and a graph, find the lowest color not used by
any neighboring node
so we can color a graph.
-}
lowColor : comparable -> Graph comparable -> Int
lowColor n ( nodes, edges ) =
    let
        values =
            Set.toList
            <| Set.remove n
            <| Set.fromList
            <| List.concatMap (\( a, b, w ) -> [ a, b ])
            <| List.filter (\( a, b, w ) -> (a == n) || (b =
= n)) edges

        neighborNodes =
            List.filter (\( v, d, c ) -> List.member v values) n
odes

        neighborColors =
            List.map (\( v, d, c ) -> c) neighborNodes
    in
        lowestMissingInt 1 neighborColors

lowestMissingInt : Int -> List Int -> Int
lowestMissingInt start list =
    if List.member start list then
        lowestMissingInt (start + 1) list
    else
        start

```

```
{-| Given a graph return a graph with the sorted from highest to
lowest degree
-}
sortByDegree : Graph comparable -> Graph comparable
sortByDegree ( nodes, edges ) =
    ( List.reverse (List.sortBy (\( n, d, c ) -> d) nodes), edge
s )

{-| Given a graph return a graph with the degree of each node co
rrectly set
    so we can color the graph with the minimum number of colors,
    set all nodes to default high color
-}
degree : Int -> Graph comparable -> Graph comparable
degree defColor ( nodes, edges ) =
    let
        endPoints =
            List.concatMap (\( a, b, w ) -> [ a, b ]) edges

        nodes_ =
            List.map (\( n, d, c ) -> ( n, (countMembers n endPo
ints), defColor )) nodes
    in
        ( nodes_, edges )

countMembers : a -> List a -> Int
countMembers value list =
    List.length <| List.filter ((==) value) list
```

[Back to problem](#)

## Problem 87a Solution

```
import Html
import List
import Set
import String

type alias Edge comparable =
  ( comparable, comparable )

type alias Graph comparable =
  ( List comparable, List (Edge comparable) )

{-| Given a graph and a starting node, return all nodes in depth
-first order.
-}
depthFirst : comparable -> Graph comparable -> List comparable
depthFirst start (nodes, edges) =
  List.reverse (depthFirst_ [start] [] (nodes, edges))

{-| Helper function for depthFirst that records visited nodes to
avoid
cyclic paths.
-}
depthFirst_ : List comparable -> List comparable -> Graph comparable -> List comparable
depthFirst_ unvisited visited ( nodes, edges ) =
  case unvisited of
    [] ->
      visited

    u::us ->
      let
```

```

        nextNodes = unique
        <| List.map (Tuple.second)
        <| (++) (List.filter \( a, b ) -> a == u) edges)

    <| List.map (\x -> ( Tuple.second x, Tuple.first x ))

    <| List.filter \( a, b ) -> b == u) edges

    newNodes = List.sort
    <| List.filter (\x -> not (List.member x (u::visited))) nextNodes
    in
    depthFirst_ (remove u (newNodes ++ unvisited)) (
u::visited) (nodes, edges)

remove : a -> List a -> List a
remove x =
    List.filter ((/=) x)

{-| Given a list, return all unique values in the list.
-}
unique : List comparable -> List comparable
unique list =
    Set.toList <| Set.fromList list

```

[Back to problem](#)



## Problem 87b Solution

```

{-| Given a graph and a starting node, return all nodes in breadth-first order.
-}
breadthFirst : comparable -> Graph comparable -> List comparable
breadthFirst start (nodes, edges) =
    List.reverse (breadthFirst_ [start] [] (nodes, edges))

{-| Helper function for breadthFirst that records visited nodes
    to avoid
        cyclic paths.
-}
breadthFirst_ : List comparable -> List comparable -> Graph comparable -> List comparable
breadthFirst_ unvisited visited ( nodes, edges ) =
    case unvisited of
        [] ->
            visited

        u::us ->
            let
                nextNodes = unique
                    <| List.map (Tuple.second)
                    <| (++) (List.filter (\( a, b ) -> a == u) edges)
                    <| List.map (\x -> ( Tuple.second x, Tuple.first x ))
                    <| List.filter (\( a, b ) -> b == u) edges

                newNodes = List.sort
                    <| List.filter (\x -> not (List.member x (u::visited))) nextNodes
            in
                breadthFirst_ (remove u (newNodes ++ unvisited))
                    (u::visited) (nodes, edges)

```

```
remove : a -> List a -> List a
remove x =
  List.filter ((/=) x)
```

```
{-| Given a list, return all unique values in the list.
-}
```

```
unique : List comparable -> List comparable
unique list =
  Set.toList <| Set.fromList list
```

[Back to problem](#)

## Problem 88 Solution

```
{-| Given a graph, return connected subsets of nodes.
-}
connected : Graph comparable -> List (List comparable)
connected (nodes, edges) =
    case nodes of
        [] ->
            []

        n::ns ->
            let
                subNodes = depthFirst n (nodes, edges)
            in
                subNodes :: connected ((diff nodes subNodes), edges)

{-| return the set of elements of a list not found in another list
-}
diff : List comparable -> List comparable -> List comparable
diff minuend subtraend =
    Set.toList <| Set.diff (Set.fromList minuend) (Set.fromList subtraend)

{-| Given a graph and a starting node, return all nodes in depth-first order.
-}
depthFirst : comparable -> Graph comparable -> List comparable
depthFirst start (nodes, edges) =
    List.reverse (depthFirst_ [start] [] (nodes, edges))

{-| Helper function for depthFirst that records visited nodes to avoid
```

```

    cyclic paths.
-}
depthFirst_ : List comparable -> List comparable -> Graph comparable -> List comparable
depthFirst_ unvisited visited ( nodes, edges ) =
    case unvisited of
        [] ->
            visited

        u::us ->
            let
                nextNodes = unique
                    <| List.map (Tuple.second)
                    <| (++) (List.filter (\( a, b ) -> a == u) edges)
                    <| List.map (\x -> ( Tuple.second x, Tuple.first x ))
                    <| List.filter (\( a, b ) -> b == u) edges

                newNodes = List.sort
                    <| List.filter (\x -> not (List.member x (u::visited))) nextNodes
            in
                depthFirst_ (remove u (newNodes ++ unvisited)) (
                    u::visited) (nodes, edges)

remove : a -> List a -> List a
remove x =
    List.filter ((/=) x)

{-| Given a list, return all unique values in the list.
-}
unique : List comparable -> List comparable
unique list =
    Set.toList <| Set.fromList list

```

[Back to problem](#)

