8 Odds and ends

The exercises below evolve around the implementation of a Huffman coding program, which provides a mechanism for compressing ASCII text. The purpose of the exercises is to allow you to apply what you have learned in "Functional Programming 1" to a slightly larger task. (However, even though the exercises share a common theme, they are fairly independent e.g. you can attempt the last exercise without having solved the earlier ones.)

The conventional representation of a textual document on a computer uses the ASCII character encoding scheme. The scheme uses seven or eight bits to represent a single character; a document containing 1024 characters will therefore occupy one kilobyte. The idea behind Huffman coding is to use the fact that some characters appear more frequently in a document. Therefore, Huffman encoding moves away from the fixed-length encoding of ASCII to a variable-length encoding in which the frequently used characters have a smaller bit encoding than rarer ones. As an example, consider the text

```
hello world
```

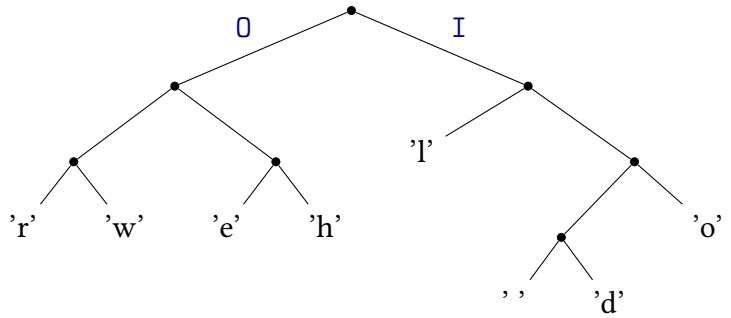
and note that the letter `l` occurs thrice. The table shown below gives a possible Huffman encoding for the eight letters:

```
'r'  OOO  'e'  OIO  'l'  IO    'd'  IIOI
'w'  OOI  'h'  OII  ' '  IIOO  'o'  III
```

The more frequent a character, the shorter the code. Using this encoding the ASCII string above is Huffman encoded to the following data:

```
OIIOIOIOIOIIIIOOOOIIIIOOOIOIIOI
```

It is important that the codes are chosen in such a way that an encoded document gives a unique decoding that is the same as the original document. The reason why the codes shown above are decipherable is that *no code is a prefix of any other code*. This property is easy to verify if the code table is converted to a code tree:



Each code corresponds to a path in the tree e.g. starting at the root the code `OII` guides us to `'h'` i.e. we walk left, right, and right again. Given a code tree and an encoded document, the original text can be decoded.

For a slightly larger example, consider the text shown in Table 3. The document is 696 characters long, but contains only 35 different characters (including the newline character `'n'`). The shortest Huffman code for this document is 3 bits long (for `' '`); the longest codes comprise 10 bits (for `'A'`, `'C'`, `'F'`, `'I'`, `'S'`, `'W'`, `'x'`, and `'z'`). The Huffman encoded document has a total length

```
As software becomes more and more complex, it
is more and more important to structure it
well. Well-structured software is easy to
write, easy to debug, and provides a
collection of modules that can be re-used to
reduce future programming costs. Conventional
languages place a conceptual limit on the way
problems can be modularised. Functional
languages push those limits back. In this
paper we show that two features of functional
languages in particular, higher-order
functions and lazy evaluation, can contribute
greatly to modularity. Since modularity is
the key to successful programming, functional
languages are vitally important to the real
world.
```

Table 3: Excerpt of “Why Functional Programming Matters” by J. Hughes

of 3019 bits. As the original text is $8 \times 696 = 5568$ bits long, we obtain a compression factor of almost 2. (Of course, this is not quite true as in reality we also need to store the code tree along the compressed data.)

Strange results are obtained if the encoded document contains just one character, repeated many times—for which the Huffman tree consists only of a single leaf, and the path describing it is empty. Therefore, we assume that the to-be-encoded text contains at least *two* different characters.

Exercise 8.1 (Warm-up: constructing a frequency table, `Huffman.lhs`). Define a function

```
frequencies  ::  (Ord char) ⇒ [char] → [With Int char]
```

that constructs a frequency table, a list of frequency-character pairs, that records the number of occurrences of each character within a piece of text. For example,

```
⋙ frequencies "hello world"
[1 :- ' ',1 :- 'd',1 :- 'e',1 :- 'h',3 :- 'l',2 :- 'o',1 :- 'r',1 :- 'w']
```

The datatype `With a b` is similar to the pair type `(a, b)`, but with a twist: when comparing two pairs only the first component is taken into account:

```
infix 1 :-
data With a b  =  a :- b

instance (Eq a) ⇒ Eq (With a b) where
  (a :- _) == (b :- _)  =  a == b
instance (Ord a) ⇒ Ord (With a b) where
  (a :- _) ≤ (b :- _)  =  a ≤ b
```

We use `With a b` instead of `(a, b)` as this slightly simplifies the next step.

Exercise 8.2 (Constructing a Huffman tree, `Huffman.lhs`). A Huffman tree or simply a code tree is an instance of a *leaf tree*, a tree in which all data is held in the leaves. Such a tree can be defined by the datatype declaration

```
data Tree elem  =  Leaf elem | Tree elem :^: Tree elem
```

The algorithm for constructing a Huffman tree works as follows:

- sort the list of frequencies on the *frequency* part of the pair –i.e. less frequent characters will be at the front of the sorted list;
- convert the list of frequency-character pairs into a list of frequency-tree pairs, by mapping each character to a leaf;
- take the first two pairs off the list, add the frequencies and combine the trees to form a branch; insert this pair into the remaining list of pairs in such a way that the resulting list is still sorted on the frequency part;
- repeat the previous step until a singleton list remains, which contains the Huffman tree for the character-frequency pairs.

(Do you see why the special pair type `With Int char` is useful?)

For example, for the sorted list of frequencies

```
[1 :- ' ',1 :- 'd',1 :- 'e',1 :- 'h',1 :- 'r',1 :- 'w',2 :- 'o',3 :- 'l']
```

the algorithm takes the following steps (`L` is shorthand for `Leaf`):

```
[1 :- L ' ',1 :- L 'd',1 :- L 'e',1 :- L 'h',1 :- L 'r',1 :- L 'w',2 :- L 'o',3 :- L 'l']
[1 :- L 'e',1 :- L 'h',1 :- L 'r',1 :- L 'w',2 :- (L ' ' :^: L 'd'),2 :- L 'o',3 :- L 'l']
[1 :- L 'r',1 :- L 'w',2 :- (L 'e' :^: L 'h'),2 :- (L ' ' :^: L 'd'),2 :- L 'o',3 :- L 'l']
[2 :- (L 'r' :^: L 'w'),2 :- (L 'e' :^: L 'h'),2 :- (L ' ' :^: L 'd'),2 :- L 'o',3 :- L 'l']
```

```
[2 :- (L ' ' :^: L 'd'),2 :- L 'o',3 :- L 'l',4 :- ((L 'r' :^: L 'w') :^: (L 'e' :^: L 'h'))]
[3 :- L 'l',4 :- ((L ' ' :^: L 'd') :^: L 'o'),4 :- ((L 'r' :^: L 'w') :^: (L 'e' :^: L 'h'))]
[4 :- ((L 'r' :^: L 'w') :^: (L 'e' :^: L 'h')),7 :- (L 'l' :^: ((L ' ' :^: L 'd') :^: L 'o'))]
[11 :- (((L 'r' :^: L 'w') :^: (L 'e' :^: L 'h')) :^: (L 'l' :^: ((L ' ' :^: L 'd') :^: L 'o')))]
```

First, the characters that occur only once are combined. The most frequent character, `'l'`, is considered only in the second, but last step, which is why it ends up high in the tree. The final tree is the Haskell rendering of the code tree shown in the introduction above.

1. Write a function

   ```
   huffman :: [With Int char] → Tree char
   ```

   that constructs a code tree from a frequency table. For example,

   ```
    ⋙ huffman (frequencies "hello world")
   ((Leaf 'r' :^: Leaf 'w') :^: (Leaf 'e' :^: Leaf 'h'))
     :^: (Leaf 'l' :^: ((Leaf ' ' :^: Leaf 'd') :^: Leaf 'o'))
   ```

   yields the Huffman tree shown above.

2. Apply the algorithm to the relative frequencies of letters in the English language, see for example `https://en.wikipedia.org/wiki/Letter_frequency`.

3. *Optional:* lists are the functional programmers favourite data structure but they are not always appropriate. The algorithm above uses an ordered list to maintain frequency-tree pairs. A moment's reflection reveals that the ordered list really serves as a *priority queue*, where priorities are given by frequencies. The central step of the algorithm involves extracting two pairs with minimum frequency and inserting a freshly created pair. Both of these operations are well supported by priority queues. If you feel energetic, implement a priority queue and use the implementation to replace the type of ordered lists.

Exercise 8.3 (Encoding ASCII text, `Huffman.lhs`). It is now possible to Huffman encode a document represented as a list of characters.

1. Write a function

   ```
   data Bit = O | I
   encode :: (Eq char) ⇒ Tree char → [char] → [Bit]
   ```

   that, given a code tree, converts the list of characters into a sequence of bits representing the Huffman coding of the document. For example,

   ```
    ⋙ ct = huffman (frequencies "hello world")
    ⋙ encode ct "hello world"
   [O,I,I,O,I,O,I,O,I,O,I,I,I,I,I,O,O,O,O,I,I,I,I,O,O,O,I,O,I,I,O,I]
   ```

   Test your function on appropriate test data, cutting and pasting the example evaluations into your file. *Hint:* it is useful to first implement a function

   ```
   codes :: Tree char → [(char, [Bit])]
   ```

that creates a code table, a character-code list, from the given code tree. This greatly simplifies the task of mapping each character to its corresponding Huffman code. For example,

```
 ⋙ codes ct
[('r',[0,0,0]),('w',[0,0,1]),('e',[0,1,0]),('h',[0,1,1]),
   ('l',[1,0]),(' ',[1,1,0,0]),('d',[1,1,0,1]),('o',[1,1,1])]
```

2. *Optional:* lists are the functional programmers favourite data structure but they are not always appropriate. The function `codes` should really yield a *finite map* (aka dictionary or look-up table), which maps characters to codes. An association list, a list of key-value pairs, is a simple implementation of a finite map. Better implementations include balanced search trees or tries. If you feel energetic, implement a finite map and use the implementation to replace the type of association lists.

Exercise 8.4 (Decoding a Huffman binary, `Huffman.lhs`). Finally, write a function

```
decode :: Tree char → [Bit] → [char]
```

that, given a code tree, converts a Huffman-encoded document back to the original list of characters. For example,

```
 ⋙ ct = huffman (frequencies "hello world")
 ⋙ encode ct "hello world"
[0,1,1,0,1,0,1,0,1,0,1,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,1,0,1,1,0,1]
 ⋙ decode ct it
"hello world"
```

Again, test your function on appropriate test data, cutting and pasting the example evaluations into your file. In general, decoding is the inverse of encoding: `decode ct ∘ encode ct = id`. Use this relationship to test your program more thoroughly.

9 Lazy evaluation

Exercise 9.1 (Pencil and paper). Recall the functional implementation of Insertion Sort from the very first lecture of "Functional Programming 1".

```
insertionSort :: (Ord a) ⇒ [a] → [a]
insertionSort []       =  []
insertionSort (x : xs)  =  insert x (insertionSort xs)

insert :: (Ord a) ⇒ a → [a] → [a]
insert a []    =  [a]
insert a (b : xs)
  | a ≤ b      =  a : b : xs
  | otherwise  =  b : insert a xs
```

Harry Hacker proposes to implement the function `minimum` in terms of `insertionSort`:

```
minimum  ::  (Ord a) ⇒ [a] → a
minimum  =  Prelude.head ∘ insertionSort
```

1. Evaluate the expression `minimum [2, 7, 1, 9, 6, 5]` by hand—twice, first using the applicative-order evaluation strategy and then using the normal-order strategy. (There is no need to meticulously list all the steps; only show the major ones.)

2. What's the running-time of `minimum` in a strict language, one that uses applicative-order evaluation? What's the running-time in a lazy language such as Haskell?

Exercise 9.2 (Dynamic programming, `Chain.lhs`). Recall that an operation, say, '`#`' is associative iff `(a # b) # c = a # (b # c)`. Associativity allows us to chain operations without introducing ambiguity e.g. `a # b # c # d`. There are 5 different ways to parenthesize the expression, but thanks to associativity all of them are equal. However, while the final results are the same, the computations may differ widely in terms of costs: space and time consumption.

For example, recall that the running-time of list concatenation `++` is proportional to the length of its first argument. Thus, if a_i is a list of length `i`, then the computation on the left is almost twice as expensive as the one on the right:

$(((a_1 ++ a_2) ++ a_3) ++ a_4$	$a_1 ++ (a_2 ++ (a_3 ++ a_4))$
`1 + 3 + 6 = 10`	`1 + 2 + 3 = 6`

Another example is afforded by matrix multiplication.[8] The cost of multiplying an `i * j` matrix by an `j * k` matrix is roughly `i * j * k`. Say $A_{(i,j)}$ is a matrix of dimension `i * j`, then there are two ways to compute the chain $A_{(1,2)} * A_{(2,3)} * A_{(3,4)}$:

$(A_{(1,2)} * A_{(2,3)}) * A_{(3,4)}$	$A_{(1,2)} * (A_{(2,3)} * A_{(3,4)})$
`1*2*3 + 1*3*4 = 18`	`1*2*4 + 2*3*4 = 32`

[8] Strictly speaking, matrices do not form a monoid as matrix multiplication is partial: it is only defined if the dimensions of the argument matrices match. We gloss over this detail. One can turn matrix multiplication into a total operation if the dimension is integrated into the matrix type. Matrices then form what is known as a category, a typed version of monads.

Clearly, the arrangement of the left is preferable.

How to compute the arrangement with minimum cost? To start with, we define a cost function, an *abstract* version of matrix multiplication that, given the dimensions of the argument matrices, computes the cost of the multiplication and the dimension of the resulting matrix (the type `With` was introduced last week):

```
type Cost  =  Integer
type Dim   =  (Integer, Integer)

(*) :: Dim → Dim → With Cost Dim
(i, j) * (j', k) | j == j'  =  (i * j * k) :- (i, k)
```

We additionally "lift" the abstract operation to arguments annotated with costs, accumulating the various costs:

```
(<*>) :: With Cost Dim → With Cost Dim → With Cost Dim
(c1 :- d1) <*> (c2 :- d2) = (c1 + c + c2) :- d where c :- d = d1 * d2
```

To determine the minimum cost for chaining a sequence of matrices, represented by a non-empty list of dimensions, we take a brute-force approach:

```
minCost :: [Dim] → With Cost Dim
minCost [a]  =  0 :- a
minCost as   =  minimum [ minCost bs <*> minCost cs | (bs, cs) ← split as ]
```

where the helper function `split` returns all possible ways to partition the non-empty argument list into two non-empty lists:

```
split :: [a] → [([a], [a])]
split [a]       =  []
split (a : as)  =  ([a], as) : [ (a : bs, cs) | (bs, cs) ← split as]
```

To illustrate, here are some example calls:

```
 ⋙ minCost [ (i, i + 1) | i ← [1 .. 3] ]
18 :- (1,4)
 ⋙ minCost [ (i, i + 1) | i ← [1 .. 9] ]
328 :- (1, 10)
 ⋙ minCost [(10, 30), (30, 5), (5, 60)]
4500 :- (10,60)
```

1. Abstract away from the specifics of matrix multiplication to solve the problem of optimal chains for arbitrary associative operations.

   ```
   minimumCost :: (size → size → With Cost size) → [size] → With Cost size
   ```

   The generic version is parametrized by a cost function, an abstract version of the associative operation, that computes the cost for some notion of size. Test your implementation by instantiating the generic optimizer to the problem of matrix chain multiplication.

2. Define cost functions for other associative operations:

   (a) list concatenation: the size is given by the length of the list, the cost is proportional to the size of the first argument;

(b) addition of infinite precision integers: the size is given by the number of digits, the cost is proportional to the maximum of the argument sizes. (Why?)

3. As it stands the generic optimizer is pretty useless: we only get to know the minimum cost, but not the associated expression tree. Augment the optimizer to also return the optimal expression tree (the type `Tree` was defined in the Huffman code-tree exercise of last week:

```
optimalChain ::  (size → size → With Cost size)
              → [size] → With Cost (With size (Tree size))
```

For example, applied to the cost function for matrix multiplication, we obtain

```
⋙ optimalChain (*) [(10, 30), (30, 5), (5, 60)]
4500 :- ((10,60) :- ((Leaf (10,30) :^: Leaf (30,5)) :^: Leaf (5,60)))
```

The expression tree shows that the cheapest evaluation first multiplies the matrices of dimensions `(10,30)` and `(30,5)` and then combines the result with the third matrix of dimension `(5,60)`.

Hint: try to re-use as much as possible from the previous code. Perhaps you can define `optimalChain` in terms of `minimumCost`?

4. Test your implementation using the cost function for Haskell's list concatenation. What do you observe? Why is `++` declared to be right associative: `infixr 5 ++`?

5. You may have noticed that the brute-force algorithm is rather slooow if applied to larger argument lists. The reason is simple: the number of possible arrangements grows exponentially. (You may remember that the number of binary trees of size `n` is given by the Catalan numbers.) There are, however, only a quadratic number of different sub-trees as a sub-tree corresponds to a segment of the to-be-multiplied list of elements. Use memoization to improve the running-time of the brute-force algorithm. *Hint:* a segment of a list can be represented by two integers, the index of the first and the index of the last element.

Exercise 9.3 (Infinite data structures, `Stream.lhs`). *Note:* many of the definitions below clash with definitions in the standard prelude. The skeleton file for this exercise uses a `hiding` clause to avoid these clashes, see also the appendix.

Haskell's list datatype comprises both finite and infinite lists. The type of streams defined below only contains infinite sequences.

```
data Stream elem  =  Cons { head :: elem, tail :: Stream elem }

infixr 5 <<
(<<) :: elem → Stream elem → Stream elem
a << s = Cons a s
```

As a simple example, the sequence of natural numbers is given by `from 0` where `from` is defined:

```
from :: Integer → Stream Integer
from n = n << from (n + 1)
```

1. Define functions

```
repeat  ::  a → Stream a
map     ::  (a → b) → (Stream a → Stream b)
zip     ::  (a → b → c) → (Stream a → Stream b → Stream c)
```

   that lift elements, unary operations, and binary operations pointwise to streams. For example,

```
 ≫ repeat 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...]
 ≫ map (2 *) (from 0)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, ...]
 ≫ zip (*) (from 0) (from 1)
[0, 2, 6, 12, 20, 30, 42, 56, 72, 90, 110, 132, 156, 182, 210, 240, ...]
```

2. Make `Stream elem` an instance of the `Num` type class. The general idea is that the methods are lifted pointwise to streams. This instance allows us to define the natural numbers and the Fibonacci numbers as follows:

```
nat, fib :: Stream Integer
nat  =  0 << nat + 1
fib  =  0 << 1 << fib + tail fib
```

3. Define a function

```
take :: Integer → Stream elem → [elem]
```

   that allows us to inspect a finite portion of a stream: `take n s` returns the first `n` elements of `s`. Use the function to turn `Stream elem` into an instance of `Show`.

4. The function `diff` computes the difference of a stream.

```
diff :: (Num elem) ⇒ Stream elem → Stream elem
diff s  =  tail s - s
```

   Here are some examples calls:

```
 ≫ diff fib
[1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...]
 ≫ (nat - 2) * (nat + 3)
[-6, -4, 0, 6, 14, 24, 36, 50, 66, 84, 104, 126, 150, 176, 204, 234, ...]
 ≫ diff it
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, ...]
 ≫ diff it
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
```

   The second difference of `(nat - 2) * (nat + 3)` is a constant stream, as the original stream is a polynomial of degree `2`. Finite difference has a right-inverse: anti-difference or summation. Derive its definition from the specification `diff (sum s) = s` additionally setting `head (sum s) = 0`. Here are some examples calls:

```
 ⋙ sum (2 * nat + 1)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, ...]
 ⋙ sum (3 * nat^2 + 3 * nat + 1)
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375, ...]
 ⋙ sum fib
[0, 0, 1, 2, 4, 7, 12, 20, 33, 54, 88, 143, 232, 376, 609, 986, ...]
```

Can you express the resulting streams without summation e.g. `sum (2 * nat + 1) = nat^2`?

The limit is hardcoded to:

```
limit :: Integer
limit = 16
```

Then we can give hack an instance of `Show`:

```
instance (Show a) ⇒ Show (Stream a) where
showsPrec _d = showStreamS limit

sum s = t where t = 0 << s + t
```

Exercise 9.4 (Worked example: minimax algorithm, `Minimax.lhs`). The purpose of this exercise is to explore the AI behind "strategic games", where a human plays against a computer. For simplicity, we confine ourselves to impartial two-player games where players take alternate moves and the same moves are available to both players. As an example, consider the following game:

A position of the game is given by two positive integers, say, `m, n > 0`. A move consists of picking a number, say, `m` and dividing it into two positive natural numbers `i, j > 0` such that `i + j = m`. If a player cannot make a move, i.e. `m = n = 1`, they lose. Here is an example run of the game:

initial position `5, 7`
Player A selects `7` and returns `3, 4`
Player B selects `4` and returns `1, 3`
Player A selects `3` and returns `1, 2`
Player B selects `2` and returns `1, 1`
Player A loses

The position `5, 7` is actually a losing position: no matter what Player A's first move, they will always lose, unless Player B makes a mistake. This can be seen by inspecting the entire game tree spawned by the initial position, see Figure 2.

1. Implement a function

   ```
   type Position = (Integer, Integer)

   moves :: Position → [Position]
   ```

   that returns a list of all possible moves from a given position e.g. `moves (1, 1) = []` and `moves (5,7) = [(1,4),(2,3),(1,6),(2,5),(3,4)]`.

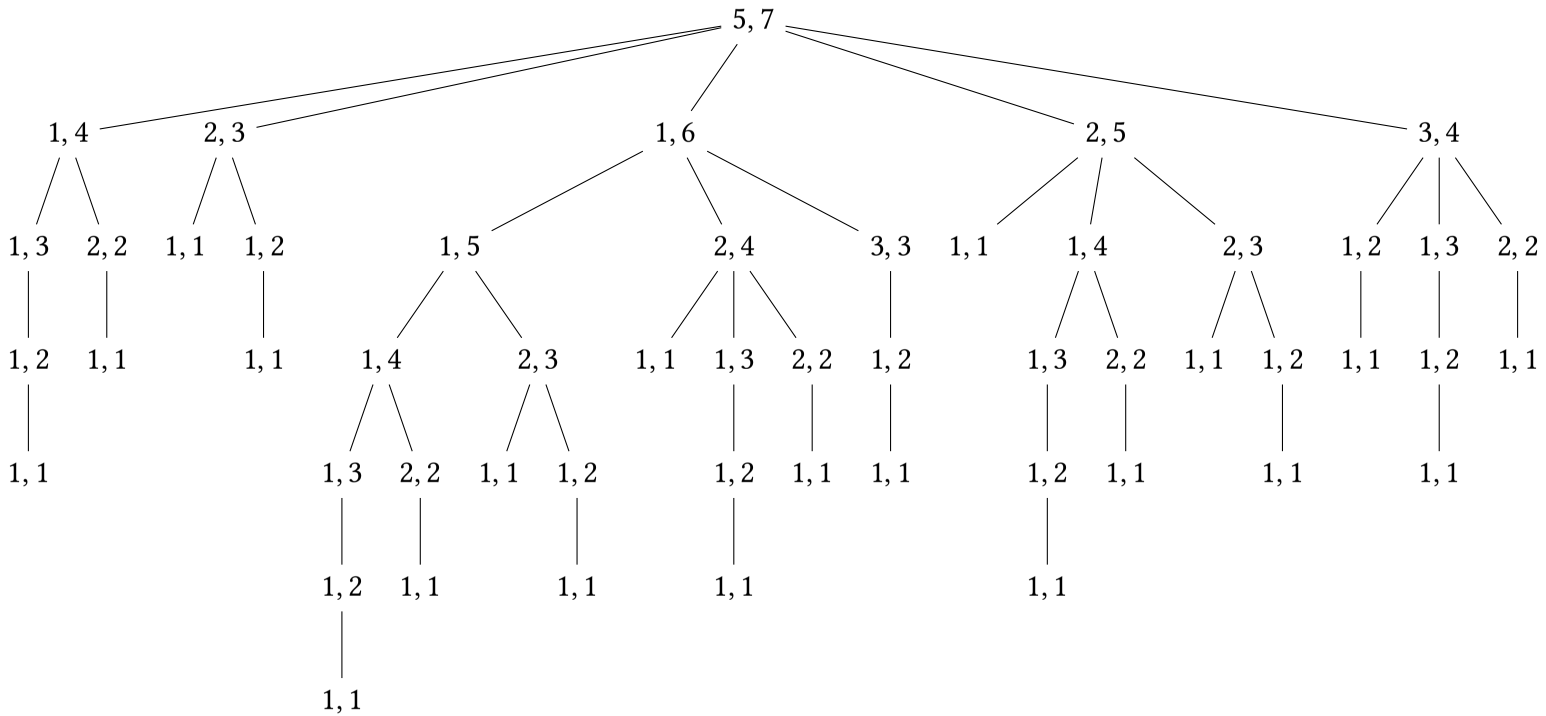


Figure 2: Game tree spawed by the initial position 5, 7 (pick'n'divide game)

2. A game tree is an instance of a *multiway tree*, which is defined by the following datatype declaration:

   ```
   data Tree elem  =  Node elem [Tree elem]
   ```

   Define a function

   ```
   gametree :: (position → [position]) → (position → Tree position)
   ```

   that constructs the entire game tree for a given initial position e.g. `gametree moves (5, 7)` should yield the tree shown in Figure 2. (Perhaps, you want to write a function that displays a game tree in a human-readable way.)

3. Implement a function

   ```
   size :: Tree elem → Integer
   ```

   that computes the size of a multiway tree. As usual, the size corresponds to the number of elements contained in the tree.

4. Define functions

   ```
   winning  :: Tree position → Bool
   losing   :: Tree position → Bool
   ```

   that determine whether the root of a game tree is labelled with a winning or a losing position. A position is a winning position iff *some* successor position is a losing position. Dually, a position is a losing position iff *all* successor positions are winning positions. By that token, a final position is a losing position.

   Apply both `size` and `winning` to some largish game tree and measure the running times (using e.g. `set +s` within GHCi). What do you observe?

5. *Optional:* show that you lose if you don't win and vice versa:

   ```
   not (winning gametree) = losing gametree
   not (losing gametree) = winning gametree
   ```

6. *Optional:* in case you need some food for thought here are some additional games worth exploring.

   mark game A position is given by a positive integer `n > 0`; the final position is `1`; a move from a non-final position consists of replacing `n` by either `n - 1` or by `n / 2` provided `n` is even.

   down-mark game The set-up is identical to the mark game, except that `n` is replaced by either `n - 1` or by $\lfloor n/2 \rfloor$.

   up-mark game Again, the set-up is identical to the mark game, except that `n` is replaced by either `n - 1` or by $\lceil n/2 \rceil$.

Exploring an entire game tree is usually not an option, in particular, if the tree is wide and deep. To illustrate, if each position gives rise to exactly two moves, then a game tree of depth 500 has $2^{500} \approx 3 * 10^{150}$ leaves. For comparison, the number of atoms in the observable universe is estimated at 10^{78}–10^{82}. Because of the exponential growth game trees are only evaluated up to some given depth e.g.

```
evaluate :: Integer → Position → Value
evaluate depth  =  maximize static ∘ prune depth ∘ gametree moves
```

7. Define a function

   ```
   prune :: Integer → Tree elem → Tree elem
   ```

   that prunes a multiway tree to a given depth i.e. the depth of `prune depth gametree` is at most `depth`.

8. Implement a function

   ```
   type Value = Int  -- [-100 .. 100]

   static :: Position → Value
   ```

   that statically evaluates a given position, returning some integer in the range `[-100 .. 100]`. In general, the higher the value, the higher we *estimate* our chances of winning. Thus, `-100` indicates a definite losing position, `100` a definite winning position. The other values reflect the uncertainty in our judgement.

   (If you have toyed a bit with the pick'n'divide game, you may be able to come up with a static evaluation that precisely predicts the outcome. In that case, you may want to consider one of the other games suggested above.)

9. Define functions

   ```
   maximize  :: (position → Value) → (Tree position → Value)
   minimize  :: (position → Value) → (Tree position → Value)
   ```

   that evaluate a game tree, based on the static evaluation function provided. The function `maximize` works as follows: if the node has no sub-trees (as a result of pruning or because it is a final position), then the static evaluation is used; otherwise, `minimize` is applied to each of the sub-trees and the maximal value of these is returned. Dually, `minimize` returns the negation of the static evaluation for leaf nodes; otherwise, it applies `maximize` to each of the subtrees, returning the minimal resulting value.

10. *Optional:* show that

    ```
    negate (maximize static gametree) = minimize static gametree
    negate (minimize static gametree) = maximize static gametree
    ```

11. *Optional:* re-implement `maximize` and `minimize` using *alpha-beta pruning*.