```
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam
nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam
erat, sed diam voluptua. At vero eos et accusam et justo duo dolores
et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est
Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur
sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et
dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea
takimata sanctus est Lorem ipsum dolor sit amet.
```

Table 1: Example text.

3 Lists

Exercise 3.1 (Programming: transformational programming, `WordList.hs`). Write a *non-recursive* program that computes the word list of a given text, ordered by frequency of occurrence.

```
type Word  =  String
wordList :: String → [(Word, Int)]
```

For clarity, `wordList` includes the frequencies in the result, e.g.

```
⋙ wordList lorem
[("accusam",2),("aliquyam",2),("at",2),("clita",2),("consetetur",2),
("dolore",2),("dolores",2),("duo",2),("ea",2),("eirmod",2),("elitr",2),
("eos",2),("erat",2),("est",2),("gubergren",2),("invidunt",2),("justo",2),
("kasd",2),("labore",2),("magna",2),("no",2),("nonumy",2),("rebum",2),
("sadipscing",2),("sanctus",2),("sea",2),("stet",2),("takimata",2),
("tempor",2),("ut",2),("vero",2),("voluptua",2),("amet",4),("diam",4),
("dolor",4),("ipsum",4),("lorem",4),("sed",4),("sit",4),("et",8)]
```

where `lorem :: String` is bound to the text displayed in Table 1.

You may find the following library functions useful (in alphabetical order): `filter`, `group`, `head`, `length`, `map`, `sort`, `sortOn`, `words`. To be able to use (some of) them you need to import `Data.List`, see Appendix **??** for details. (As an aside, `sort` and `sortOn` implement stable sorting algorithms. Why is this a welcome feature for this particular application?) Can you format the output so that one entry is shown per line?

Exercise 3.2 (Warm-up: list design pattern). Using the *list design pattern* discussed in the lectures, give recursive definitions of

1. a function `allTrue :: [Bool] → Bool` that determines whether every element of a list of Booleans is true;

2. a function `allFalse` that similarly determines whether every element of a list of Booleans is false;

3. a function `member :: (Eq a) ⇒ a → [a] → Bool` that determines whether a specified element is contained in a given list;

4. a function `smallest :: [Int] → Int` that calculates the smallest value in a list of integers;

5. a function `largest` that similarly calculates the largest value in a list of integers.

The purpose of this exercise is to train the list design pattern for defining list consumers. However, once we have covered the corresponding material in the lectures, you may want to return to consider which of these functions can be written more simply using standard *higher-order functions* like `map` and `foldr`.

Exercise 3.3 (Programming). A *run* is a non-empty, non-decreasing sequence of elements. Use the *list design pattern* to define a function

```
runs :: (Ord a) ⇒ [a] → [[a]]
```

that returns a list of runs such that `concat ∘ runs = id`, e.g.

```
 ⋙ runs "hello, world!\n"
["h", "ello", ",", " w", "or", "l", "d", "!", "\n"]
 ⋙ concat it
"hello, world!\n"
```

Partitioning a list into a list of runs is a useful pre-processing step prior to sorting a list. Do you see why?

Exercise 3.4 (Programming, `DNA.hs`). Recall the representation of bases and DNA strands introduced in the lectures.

```
data Base  =  A | C | G | T
deriving (Eq, Ord, Show)

type DNA      =  [Base]
type Segment  =  [Base]
```

1. Define a function `contains :: Segment → DNA → Bool` that checks whether a specified DNA segment is contained in a DNA strand. Can you modify the definition so that a list of positions of occurrences is returned instead?

2. Define a function `longestOnlyAs :: DNA → Integer` that computes (the length of) the longest segment that contains only the base `A`.

3. Define a function `longestAtMostTenAs :: DNA → Integer` that computes (the length of) the longest segment that contains at most ten occurrences of the base `A`. (This is more challenging. Don't spend too much time on this part.)

Exercise 3.5 (Worked example: testing, `QuickTest.hs`).

> *Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.*
>
> The Humble Programmer by Edsger W. Dijkstra

Beware of bugs in the above code;
I have only proved it correct, not tried it.
Donald Knuth

Harry Hacker has implemented a very nifty, highly efficient sorting function. At least, that's what he thinks. We are probably well advised to test his program thoroughly before using it in production code. The purpose of this exercise is to develop a little library for testing programs. The library can be seen as a tiny *domain-specific language* (DSL) for testing embedded in Haskell. (More on DSLs and EDSLs in §5.)

The principal idea is to take a *type-driven* approach to testing. To scrutinize a function `f` of type, say, `A → B` we need probes for inputs of type `A`, and the to-be-verified property for results of type `B`. For simplicity, probes are just lists of inputs and a property is just a Boolean function.

```
type Probes a    =  [a]

type Property a  =  a → Bool
```

Given `proba :: Probes A`, and `propb :: Property B`, the expression `proba ?→ propb` is then a test procedure for functions of type `A → B`. Applied to `f` i.e. `(proba ?→ propb) f` it exercises its argument with all possible probes, checking each of the resulting outputs. For example, Harry's sorting function is exercised by

```
(permutations [0 .. 9] ?→ ordered) niftySort
```

Here `permutations` generates all permutations of its input list and `ordered` checks whether a list is ordered. Of course, Harry could easily defeat the test by defining `niftySort xs = []`. A better approach is to check his implementation against a *trusted* sorting algorithm. This is accomplished using a variant of `?→`:

```
(permutations [0 .. 9] ?⇒ \ inp res → trustedSort inp == res) niftySort
```

Here, `inp` is bound to the original input and `res` is the result of Harry's program.

Turning to the implementation, `?→` and `?⇒` can be conveniently implemented using list comprehensions.

```
infixr 1  ?→, ?⇒

(?→)   :: Probes a → Property b → Property (a → b)
(?⇒)   :: Probes a → (a → Property b) → Property (a → b)

probes ?→ prop  =  \ f → and [ prop (f x)    | x ← probes ]
probes ?⇒ prop  =  \ f → and [ prop x (f x) | x ← probes ]
```

1. Define the predicate

   ```
   ordered :: (Ord a) ⇒ Property [a]
   ```

   that checks whether a list is ordered i.e. the sequence of elements is non-decreasing.

2. Apply the list design pattern to define the generator

   ```
   permutations :: [a] → Probes [a]
   ```

that produces the list of all permutations of its input list. (How many permutations of a list of length `n` are there?)

3. Use the combinators to define a testing procedure for the function

   ```
   runs :: (Ord a) ⇒ [a] → [[a]]
   ```

   of Exercise 3.3.

4. Harry Hacker has translated a function that calculates the *integer square root* from C to Haskell.

   ```
   isqrt :: Integer → Integer
   isqrt n = loop 0 3 1
     where loop i k s  | s ≤ n      = loop (i + 1) (k + 2) (s + k)
                       | otherwise  = i
   ```

   It is not immediately obvious that this definition is correct. Define a testing procedure `isIntegerSqrt :: Property (Integer → Integer)` to exercise the program. Can you actually figure out how it works?

5. Define a combinator

   ```
   infixr 4  comb
   (comb)     :: Probes a → Probes b → Probes (a, b)
   ```

   that takes probes for type `a`, probes for type `b`, and generates probes for type `(a, b)` by combining the input data in all possible ways e.g.

   ```
    ⋙ bools = [False, True]
    ⋙ bools comb bools
   [(False,False),(False,True),(True,False),(True,True)]
    ⋙ bools comb bools comb bools
   [(False,(False,False)),(False,(False,True)),(False,(True,False)),(False,(True,True)),
   (True,(False,False)),(True,(False,True)),(True,(True,False)),(True,(True,True))]
    ⋙ chars = "Ralf"
    ⋙ bools comb chars
   [(False,'R'),(False,'a'),(False,'l'),(False,'f'),(True,'R'),
   (True,'a'),(True,'l'),(True,'f')]
   ```

   If `as` contains `m` elements, and `bs` contains `n` elements, then `as comb bs` contains …

Exercise 3.6 (Algorithmics: greedy algorithms, `Format.hs`). An important problem in text processing is to format text into lines of some fixed width, ensuring as many words as possible on each line. If we assume that adjacent words are separated by one space, a list of words `ws` will fit in a line of width `n` if `length (unwords ws) ≤ n`. Define a function

```
format :: Int → [Word] → [[Word]]
```

that given a maximal line width and a list of words returns a list of fitting lines so that `concat ∘ format n = id`. Is this always possible? (As an aside, a function `f` with the property `concat ∘ f =id` computes a *partition* of its input. Have you seen this property before?)

For example, to format the text shown in Table 1 to a line width of `40` we type:

```
⋙ putStr $ unlines $ map unwords $ format 40 $ words lorem
```

The resulting output is shown below

```
Lorem ipsum dolor
sit amet, consetetur sadipscing elitr,
sed diam nonumy eirmod tempor invidunt
ut labore et dolore magna aliquyam
erat, sed diam voluptua. At vero eos
et accusam et justo duo dolores et ea
rebum. Stet clita kasd gubergren, no sea
takimata sanctus est Lorem ipsum dolor
sit amet. Lorem ipsum dolor sit amet,
consetetur sadipscing elitr, sed diam
nonumy eirmod tempor invidunt ut labore
et dolore magna aliquyam erat, sed
diam voluptua. At vero eos et accusam
et justo duo dolores et ea rebum. Stet
clita kasd gubergren, no sea takimata
sanctus est Lorem ipsum dolor sit amet.
```

Apply the list design pattern to define `format`. You may want to take a greedy approach that makes locally optimal choices. For simplicity, we are content if the first line, but only the first line, wastes a lot of space. Once we have introduced the function `foldl` in the lectures, you may want to revisit this simplifying, but slightly odd assumption: usually, waste is permitted only in last line. (In general, text formatting is an optimization problem for some suitable measure of "badness" of formatting.)

Exercise 3.7 (List notation). In this exercise you practice reading list notation. Several lists are written down below. For every list do the following:

- explain in your own words which list is written down;
- give the number of elements in the list;
- give the shortest notation that results in an equal list (this could be the same expression); and
- give the result of the standard functions `head`, `tail`, `init`, and `last` on the list. You can find these functions in the standard Prelude.

These are said lists:

```
l1 = []
l2 = 1000:[]
l3 = [[]]
l4 = [[]:[]]
l5 = 'a':'b':'c':'d':'e':[]
l6 = [[[1]],[],[2,3]:[[4,5,6]]]
l7 = [[[1,2],[3,4]],[],[[],[],[5,6]]]
l8 = [1:2:3:4:5:[]]
```

Exercise 3.8 (Types and values). In this exercise you are given a number of list types. Give one expression for each type that corresponds to that type.

```
e1 :: [Int]
e1 = ...

e2 :: [Bool]
e2 = ...

e3 :: [[Int]]
e3 = ...

e4 :: [[[Real]]]
e4 = ...

e5 :: [Int → Int → Int]
e5 = ...
```

Exercise 3.9 (Type inference and lists). Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions. Please note that you can use GHCi to check your answer.

```
f1         = []

f2         = [[]]

f3 x       = [x]

f4 (x,y)   = [x,y]

f5 (x,y)   = [(x,y)]

f6 x       = [[x]]

f7 (x:y:z) = (x,y)

f8 (x:y)   = (x,y)

f9 [x]     = x
f9 [x,y]   = x
f9 (x:xs)  = f9 (init xs)
```

Exercise 3.10 (The first will be the last). The purpose of this assignment is that you practice writing recursive functions on lists yourself, using *guards* and *pattern-matching*. Therefore, do not use any of the pre-defined functions that are available in prelude.

First two and last two Write the function `first2` and `last2` that both get a *list* as argument and return a *list*. The function `first2` returns the first two elements of the argument and

`last2` returns the last two elements. Generate an error using the function `error` when the argument lists are too short.

Examples:

```
first2 [42]         =  "*** Exception:  List is too short."
first2 [1,2,3,4,5]  =  [1,2]
last2 [42]          =  "*** Exception:  List is too short."
last2 [1,2,3,4,5]   =  [4,5]
```

First *n* and last *n* Write the recursive functions `firstn` and `lastn` that both get a number and a *list* as argument and return a *list*. The type of both functions thus is: `[a] → Int → [a]`. The function `firstn` returns the first `n` elements and `lastn` returns the last `n` elements of a given list. Generate an error using the function `error` when the argument lists are too short.

Examples:

```
firstn 4 [42]         =  "*** Exception:  list is too short ...  "
firstn 4 [1,2,3,4,5]  =  [1,2,3,4]
lastn 4 [42]          =  "*** Exception:  list is too short ...  "
lastn 4 [1,2,3,4,5]   =  [2,3,4,5]
```

Properties Complete the following statements:

```
∀0 ≤ n,      xs :: [a] : firstn  n  (firstn  n  xs)  =  ...
∀0 ≤ n,      xs :: [a] : firstn  n  (lastn   n  xs)  =  ...
∀0 ≤ n,      xs :: [a] : lastn   n  (firstn  n  xs)  =  ...
∀0 ≤ n,      xs :: [a] : lastn   n  (lastn   n  xs)  =  ...
∀0 ≤ m ≤ n,  xs :: [a] : firstn  m  (firstn  n  xs)  =  ...
∀0 ≤ m ≤ n,  xs :: [a] : length     (firstn  m  xs)  ?  length (firstn n xs)
```

Exercise 3.11 (Glueing lists). The append operator (`++`) is defined as follows:

```
(++) :: [a] →  [a] → [a]
[ ]      ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Calculate the results of the expressions below by rewriting the `++` operator. The values of x_i are irrelevant. Part **6** is challenging!

1. `[] ++ []`
2. `[] ++ [x0,x1] ++ []`
3. `[[]] ++ [x0,x1]`
4. `[x0,x1] ++ [[]]`
5. `[x0,x1,x2] ++ ([x3,x4] ++ ([x5] ++ []))`
6. `(([x0,x1,x2] ++ [x3,x4]) ++ [x5]) ++ []`

Use the results of part **5** and **6** to argue why the `++` operator is *right associative*.

Exercise 3.12 (List generators). The purpose of this assignment is that you practice writing recursive functions that generate lists yourself, using *guards* and *pattern-matching*. Therefore, do not use any of the pre-defined functions that are available in the `Prelude` to create lists. Assignment 3.14 is used for that purpose.

Write the following functions that generate lists:

- `all_elements` x calculates the infinite list $[x, x, x \ldots]$.
  Example: `all_elements 42` = `[42, 42, 42, 42, ...]`.
- `step` x calculates the infinite list `[`x`,succ` x`, succ(succ` x `),...]`.
  Example: `step -10` = `[-10, -9, -8, -7 ...]`.
  Example: `step 'a'` = `['a','b','c','d' ...]`.
- `from_step` x z calculates the infinite list $[x, x + z, x + z + z, \ldots]$.
  Example: `from_step 0 -2` = `[0, -2, -4, -6 ...]`.
- `from_to` x f (with $x \leq y$) calculates the list `[`x`,succ` x`, succ(succ` x `),...,`y'`]`in such a way that $y' \leq y$ and `succ` $y' > y$.
  `from_to x y` (with $x > y$) returns the empty list.
  Example: `from_to 'a' 'z'` = `['abcdefghijklmnopqrstuvwxyz']`.
  Example: `from_to 0.5 7.0` = `[0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5]`.
  Example: `from_to 3 3` = `[3]`.
  Example: `from_to 3 0` = `[]`.
- `from_to_step` x y z calculates the list $[x, x+z, x+z+z, \ldots, y']$ in such a way that $x \leq y \wedge z > 0 : y' \leq y \wedge y' + z > y$ and if $x \geq y \wedge z < 0 : y' \geq y \wedge y' + z < y$. For all other cases the result is the empty list.
  example: `from_to_step 0 10 2` = `[0, 2, 4, 6, 8, 10]`.
  example: `from_to_step 0 -10 -3` = `[0, -3, -6, -9]`.
  example: `from_to_step 0 -10 -42` = `[0]`.
  example: `from_to_step 0 -10 2` = `[]`.

Make sure that the function types are as general as possible.

Exercise 3.13 (ZF-notations). For all the functions below explain what they calculate, and additionally give the most general type of each of the functions.

```
g1 as bs = [(a,b) | a ← as,  b ← bs]
g3 as bs = [(a,b) | a ← as,  b ← bs , a < b]
g4 as bs = [  a   | a ← as,  b ← bs , a == b]
g5   xss = [  x   | xs ← xss, x ← xs]
g6 a  xs = [  i   | i  ← [0 ..] & x ← xs , a == x]
```

Exercise 3.14 (List generators, part II). Write the same functions as in Exercise 3.12, but now only use *list comprehensions*. It is possible that the type of the functions has to be adapted. Verify and compare the results of the new implementation with the examples from Exercise 3.12.

Exercise 3.15 (!! and ??). The operator (!) infixl 9 :: [a] -> Int -> a! selects the *i*th element from a list *xs* after calling *xs* !! *i* (counted from zero). If *xs* does not contain enough elements an error message is generated.

Implement using list comprehensions the operator

```
(??) infixl 9 :: [a] → a → Int
```

that searches for the index of x in a list xs, if x is a member of xs. If there is no such value, then *xs* `??` x should return `-1`. If xs is a list containing element x, then the following must hold: xs ! (!*xs* `??` x) = x. If x is not an element of xs, then xs ! (!*xs* `??` x) results in an error message.
Examples:

```
[1,2,3,4,5,6]  ?? 3   = 2
[1,2,3,4,5,6]  ?? 10  = -1
"Hello world"  ?? 'o' = 4
```

Exercise 3.16 (ZF and removeAt). Haskell does not have a function to remove list elements at an index. Write, using list comprehensions, the function `removeAt::Int → [a] → [a]` that accomplishes this task. For example: `removeAt 2 "abc"` ⟹ `"ac"`

Exercise 3.17 (Fibonacci). The *Fibonacci* sequence $F = F_0, F_1, F_2, \cdots$ is defined as:

$$F = \begin{cases} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{cases}$$

Write the recursive function `fibonacci` as above with the meaning: `fibonacci n` = F_n. For which values is this function defined?

Test your function with incrementing values:

```
N = 46
[fibonacci i | i ← [1 .. N]]
```

Describe and explain the execution behaviour of the program.

Exercise 3.18 (Sumlists). Write the function `sumLists` with the following meaning:

`sumLists` $[a_0, ..., a_n]$ $[b_0, ..., b_n] = [a_0 + b_0, ..., a_n + b_n]$.

You need to choose your own behaviour for lists that are of unequal length. Motivate that choice. Test your function with some representative sample inputs. Write what inputs you chose and give the result.

Exercise 3.19 (The Fibonacci sequence). Another way of thinking about the fibonacci sequence F is:

```
[1,1,2,3, 5, 8,13,21,34 ...]                F
[1,2,3,5, 8,13,21,34,55 ...]           (tl F)
-------------------------------------------------
[2,3,5,8,13,21,34,55,89 ...]  sumLists F (tl F)
```

This allows you to compute the first n fibonacci numbers as follows:

```
nFibs n = take n fibs where
   fibs = [1,1:sumLists fibs (tl fibs)]
```

Describe and explain the execution behaviour of the program.

Exercise 3.20 (Fragments). The function `frags` computes of a list *xs* all *fragments*. A fragment of *xs* is defined as a part of *xs* such that it only contains consecutive elements of *xs*. The number and the position of the first element is arbitrary (but of course limited by *xs*). Note that both `[]` and *xs* itself are fragments of *xs*. For example:

$$
\begin{array}{lcl}
\texttt{frags}\ [\,] & = & [[\,]] \\
\texttt{frags}\ [x_0 \ldots x_n] & = & [[\,] \\
 & & ,[x_0] \ldots [x_n] \\
 & & ,[x_0, x_1], [x_1, x_2], \ldots [x_{n-1}, x_n] \\
 & & \quad\vdots \\
 & & ,[x_0 \ldots x_{n-1}], [x_1 \ldots x_n] \\
 & & ,[x_0 \ldots x_n] \\
 & &] \qquad\qquad (\text{for } n \geq 0)
\end{array}
$$

1. Give the most general type of `frags`.
2. If *xs* is a list of n elements, how many elements does the list (`frags` *xs*) have?
3. Implement `frags`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

Exercise 3.21 (Sublists). The function `subs` computes of a list *xs* all *sublists*. A sublist of *xs* is a list with the same elements in the same order, but in which an arbitrary number of elements is left out. Note that both `[]` and *xs* are sublists of *xs*. For example:

$$
\begin{array}{lcl}
\texttt{subs}\ [\,] & = & [[\,]] \\
\texttt{subs}\ [x_0] & = & [[\,], [x_0]] \\
\texttt{subs}\ [x_0, x_1] & = & [[\,], [x_0], [x_1], [x_0, x_1]] \\
\texttt{subs}\ [x_0, x_1, x_2] & = & [[\,], [x_0], [x_1], [x_2], [x_0, x_1], [x_0, x_2], [x_1, x_2], [x_0, x_1, x_2]]
\end{array}
$$

1. Give the most general type of `subs`.
2. If *xs* is a list of n elements, how many elements does the list (`subs` *xs*) have?
3. Implement `subs`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

Exercise 3.22 (Permutations). The function `perms` computes all *permutations* of a list *xs*. A permutation of *xs* is a list that contains the same elements, but possibly in a different order. Note that *xs* is a permutation of *xs*. For example:

$$
\begin{array}{lcl}
\texttt{perms}\ [\,] & = & [[\,]] \\
\texttt{perms}\ [x_0] & = & [[x_0]] \\
\texttt{perms}\ [x_0, x_1] & = & [[x_0, x_1], [x_1, x_0]] \\
\texttt{perms}\ [x_0, x_1, x_2] & = & [[x_0, x_1, x_2], [x_0, x_2, x_1] \\
 & & , [x_1, x_0, x_2], [x_1, x_2, x_0] \\
 & & , [x_2, x_0, x_1], [x_2, x_1, x_0] \\
 & &]
\end{array}
$$

1. Give the most general type of `perms`.
2. If *xs* is a list of *n* elements, how many elements does the list (`perms` *xs*) have?
3. Implement `perms`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

Exercise 3.23 (Partitions). The function `partitions` computes all *partitions* of a list *xs*. A partition of *xs* is a list $[A_1, \ldots, A_n]$ ($n \geq 0$), such that $A_1 \texttt{ ++ } \ldots \texttt{++ } A_n = xs$ and every A_i is non-empty. The empty list only has itself as possible partition. For example:

$$
\begin{array}{ll}
\texttt{partitions}\ [\,] & = [[[\,]]] \\
\texttt{partitions}\ [x_0] & = [[[x_0]]] \\
\texttt{partitions}\ [x_0, x_1] & = [[[x_0], [x_1]], [[x_0, x_1]]] \\
\texttt{partitions}\ [x_0, x_1, x_2] & = [[[x_0], [x_1], [x_2]], [[x_0], [x_1, x_2]], [[x_0, x_1], [x_2]], [[x_0, x_1, x_2]]]
\end{array}
$$

1. Give the most general type of `partitions`.
2. If *xs* is a list of *n* elements, how many elements does (`partitions` *xs*) have?
3. Implement `partitions`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

Exercise 3.24 (Lists and sorting). A list $L = [l_0 \ldots l_n]$ is sorted if for every $0 \leq i < j \leq n : \ l_i \leq l_j$.

Testing for sorted Use the idea from exercise 3.19 to construct a function `is_sorted` that takes a list and tests if the list is sorted. The function `is_sorted` may only inspect the list using list comprehension.

Sorting lists The list L' is the sorted list of L if L' is a *permutation* of L *and* L' is sorted. Write the function `zfsort` that takes a list and sorts it according to this definition.

Use the function `perms` from exercise 3.22, your function `is_sorted` and only a list comprehension.

Complexity What is the order of complexity of this way of sorting in terms of the length of the list? Is it a good way to sort lists?

Exercise 3.25 (The last will be the first). The function `last` that returns the last element of a list can be written in several ways. A first method is by writing a recursive function:

```
last1 :: [a] → a
last1 [x]    = x
last1 [_:xs] = last1 xs
```

Another way is by noting that the last element of the list is the same as the first element of the reversed list (you may find the function `reverse` in Haskell's standard Prelude):

```
last2 :: ([a] → a)
last2 = hd ∘ reverse
```

Compute Rewrite the following expressions by writing each step on a new line, and by underlining what you rewrite.

1. `last1 [1,2,3,4]`
2. `last2 [1,2,3,4]`

Function equality versus execution behaviour The functions `last1` and `last2` are *equal*, that is: given the same arguments they give the same result. That does not mean they are the same. Explain the difference using the results from the previous question.

Exercise 3.26 (Number sequence). Implement the function `intChars :: (Int → [Char])` that converts an `Int` into a list of its digits (in decimal representation) as characters. In the same style, write the inverse function, `charsInt :: ([Char] → Int)`, that takes a `Char` list of all digits and computes the number represented by the list.
Example: `charsInt "3210"` = 3210.

Exercise 3.27 (Uniform sets). Complete the module in `Set.hs` for working with finite sets of elements of the same type. Sets are in this implementation basically aliasses for lists, and the conversion is using `Set`. The following properties need to be valid for operations on sets:

- The operation `toSet` converts a list of elements to a set. (*hint:* it is not the same as `Set`) Sets do not contain duplicate elements, so the list that is obtained from `fromSet` may not contain duplicates.
- The predicates `isEmptySet`, `isDisjoint` and `memberOfSet` respectively test if a given set is empty (contains no elements), two sets are disjoint (have no common elements), and if a given value is a member of a given set (is contained by the set).
- The `Set` instance of `zero` gives the empty set. The empty set has no elements. The `Set` instance of `==` tests if two sets are identical (contain the same elements). Note that in sets, the order of the elements is not defined! This means that $\{1, 2\} = \{2, 1\}$. The function `numberOfElements` returns the number of elements of a set.
- There are two predicates that test a subset-relationship. The predicate `isStrictSubset` determines if the first argument is a *strict* subset of the second argument (i.e. the second argument contains elements that do not occur in the second argument, $A \subset B$). The predicate

`isSubset` determines if the first argument is a subset of the second argument (all elements of the first argument are also an element of the second argument, $A \subseteq B$). Every set is a subset of itself, but it is not a *strict* subset of itself.

- The operations `union` and `intersection` respectively compute the union ($A \cup B$) and the intersection ($A \cap B$) of two sets. The operation `without` removes all elements of the second argument from the first argument ($A \setminus B$). The function `product` computes the Cartesian product of two sets ($A \times B$).
- The function `powerSet` computes the power set (set of all subsets, $\wp(A)$) of a given set A.

Use *ZF*-expressions, `..`-expressions or recursive functions as you see fit. Try to determine with which core functions you can construct the other functions in this module (for instance: two sets are equal if they are each other's subset).

Exercise 3.28 (Stack). Complete the module `Stack.hs` providing operations on stacks. The usual properties need to hold for the operations:

- `newStack` constructs an empty stack.
- (`push` x s) places x on top of stack s, and (`pushes` $[x_0, \ldots, x_n]$ s) places x_0, x_1, …, x_n consecutively on top of the stack. Afterwards this means that x_n is at the top of the stack.
- (`pop` s) removes the top element of the stack s if possible. (`popn` n s) removes the top n elements of stack s if possible.
- (`top` s) gives the top element of stack s if possible, and a runtime error if s is empty. (`topn` n s) gives the top n elements of the stack s if possible, and a runtime error if there are not enough elements.
- (`elements` s) returns all elements of stack s, from the top to the bottom. (`count` s) returns the number of elements on the stack s.

Try to develop a small core of functions you can use to construct the other functions in this module (for example: define `pushes` using `push`).

Exercise 3.29 (Sorted List). Complete the module `SortedList.hs`, which offers operations for working with lists of which the elements are sorted. The following properties need to hold for the operations (pay attention to the complexity restrictions below!):

- (`minimum` l) and (`maximum` l) return the minimum and maximum value of l, and generate a runtime error if l is empty. The order of runtime complexity of both functions needs to be $\mathcal{O}(1)$ (constant time).
- `newSortList` constructs a new empty, sorted list.
- (`memberSort` x l) tests if value x occurs in l. The predicate (`memberSort` x `newSortList`) is thus false for all x.

- (`insertSort` x l) inserts value x in l in the right position. The predicate (`memberSort` x (`insertSort` x l)) is thus always true for all x and l.
- (`removeFirst` x l) removes the first occurrence of x from l (there may exist duplicate elements in l). (`removeAll` x l) removes all occurrences of x from l. This means the predicate (`memberSort` x (`removeAll` x l)) is always false for every x and l.
- (`elements` l) returns all elements of l. This list is sorted and may contain duplicate elements. (`count` l) returns the number of elements of l. So: `length (elements` l`) = count` l.
- (`mergeSortList` l_1 l_2) merges l_1 and l_2 into a new sorted list. The following holds: `count` l_1 `+ count` l_2 `= count (mergeSortList` l_1 l_2`)`; if (`memberSort` x l_1), then also (`memberSort` x `(mergeSortList` l_1 l_2`)`) and (`memberSort` x `(mergeSortList` l_2 l_1`)`).

Exercise 3.30 (Association list). Complete the module `AssocList.hs`. In an *associative list* of type `AssocList k a`, elements of type `a` are stored using a *key* of type `k`. For each *key*, one element is stored. The following properties need to hold:

- `newAssocList` creates an empty association list.
- (`countValues` l) returns the number of stored elements.
- (`lookupKey` k l) gives the singleton list with value v if *key-value pair* (k, v) was present in l, and an empty list otherwise.
- (`updateKey` k v l) inserts the *key-value pair* (k, v) in l if k was not already present in l, and replaces the previous value associated with key k by v otherwise.
- (`removeKey` k l) removes the *key-value pair* associated with k if present, and leaves l unmodified otherwise.

Exercise 3.31 (Cards, part II). In exercise 4.13 you created the type `Card`. In this exercise you use them to develop some functions that generate lists of cards.

1. **Card deck** Write the function `carddeck :: [Card]` that generates a complete list of all cards in a deck. Try to write as short a definition as possible.
2. **Sorting by value** Write function `sort_by_value` that sorts a deck in ascending order. First sort by value, then by suit in the order *hearts* (♥), *diamonds* (♦), *spades* (♠) and *clubs* (♣). Try to write as short a definition as possible.
3. **Sorting by suit** Write the function `sort_by_suit` that sorts a deck of cards in increasing order. First sort by suit in the same order as above, then by value. Try to write as short a definition as possible.

Exercise 3.32 (Roman numerals). The classical Romans wrote their numbers using the following symbols: M, D, C, L, X, V en I with the respective values of 1000, 500, 100, 50, 10, 5 en 1. A positive integer is noted by writing these symbols consecutively, where the highest value symbols are on

the left, and the lowest value symbols are on the right. The value of the number is the sum of the values of the individual symbols. Negative values and zero can't be represented.

Example: MDCLXVI = 1000 + 500 + 100 + 50 + 10 + 5 + 1 = 1666.
Example: MMVIII = 1000 + 1000 + 5 + 1 + 1 + 1 = 2008.

For often occurring patterns, *abbreviations* are used:

DCCCC	⇒	CM	LXXXX	⇒	XC	VIIII	⇒	IX
CCCC	⇒	CD	XXXX	⇒	XL	IIII	⇒	IV

Example: CMXCIX = (500 + 4 · 100) + (50 + 4 · 10) + (5 + 4 · 1) = 999.
Example: CDXLIV = 4 · 100 + 4 · 10 + 4 · 1 = 444.

Let the algebraic type `RD` represent Roman 'digits', and the type `Roman` a number notated by Roman numerals.

```
data RD    = M | D | C | L | X | V | I
data Roman = Roman [RD]
```

Roman → Int Write a function `roman2int :: Roman → Int` that satisfies the properties given above.

Int → Roman Write a function `int2roman :: Int → Roman` that converts a *positive* `Int` value to the *shortest* Roman representation.

Exercise 3.33 (Word Sleuth). 'Word sleuth' or 'word search' puzzles consist of letters in an $m \times n$ matrix. A word list is supplied with words that need to be found and crossed out in the grid. Punctuation is ignored and each word occurs exactly once. A word can occur horizontally (both from left to right and from right to left), vertically (from top to bottom or reversed) and diagonally (both from left to right and right to left, both from the top to bottom and reversed). After crossing out the words it is often possible to construct a sentence of the remaining letters. The example below reveals the name of a familiar programming language after crossing out the words:

```
E  C  L  L        LAZY
P  E  I  A        LIST
Y  S  F  Z        TYPE
T  A  N  Y        ZF
```

Implement an algorithm that given an $m \times n$ matrix of letters and a word list, crosses out all words as above. The algorithm needs to return the remaining letters from left to right, top to bottom.

Exercise 3.34 (Boggle words). *Boggle* (`http://en.wikipedia.org/wiki/Boggle`) is a word game that consists of sixteen dice that have letters instead of pips. The dice are rolled and then placed in a 4 × 4 matrix. Players try to extract as many words as possible. Words are formed by starting on an arbitrary letter and then moving to a neighbouring dice that is above, below,

left, right or diagonally in any direction. Words need to have at least three letters and each die may only be used once in a word. This means the longest possible word is sixteen letters. Words may occur in both singular and plural form, verbs may not be conjugated. Perfect past tense is allowed. Names, foreign words, abbreviations and composed words are not allowed.

Implement the algorithm `boggle_words` that, given a 4×4 matrix of letters and a word list generates all valid words of at least 3 letters from the given matrix.

Exercise 3.35 (Change). In this exercise you write the program for a change machine. The change machine has an infinite amount of each kind of coin and bank note. It may however not dispense more than a specified amount of k coins / bank notes at the same time. Write the function `change` that, given the value to be exchanged, the available currency, and the maximum amount of coins / bank notes to be dispensed, computes all possible solutions.

Use the following type synonyms and the type of `change`:

```
type  Amount     = Int          // a positive number
type  Currency   = Int          // a positive number
type  Currencies = [Currency]   // a non-empty list
type  Coin       = Int          // a positive number
type  K          = Int          // a positive number
type  Change     = [Coin]

change :: Amount Currencies K → [Change]
```

Example: (The solutions may be returned in a different order)

```
change 50 [100,50,20,10,5,1] 1 ⟹ [[50]]
change 50 [100,50,20,10,5,1] 2 ⟹ [[50]]
change 50 [100,50,20,10,5,1] 3 ⟹ [[50],[10,20,20]]
change 50 [100,50,20,10,5,1] 4 ⟹ [[50],[10,20,20],[5,5,20,20],[10,10,10,20]]
```

Exercise 3.36 (Dominoes). The classical game of dominoes (the so-called *double-six* variant) consists of 28 rectangular tiles, called dominoes. Each domino has two halves that each have a number of *pips*. The 28 dominoes contain all unique combinations of pips, with the number of pips between *zero* and *six* (see also figure 1).
There exist variants with other combinations, like double-*nine*, double-*twelve* and even double-*eighteen*. We call such a collection of dominoes a double-N game (with N the maximum possible number of pips on one half of a tile). A double-N game has $MAX = \frac{(N+2)(N+1)}{2}$ dominoes. With the tiles of a double N game it is possible to make a snake, i.e. a sequence of consecutive dominoes such that the short sides are connected, and the pips on connecting halves have the same number of pips (so 0 with 0, 1 with 1, etc.). A snake does not contain 'loops', which can be played during a real game of dominoes.

Write the function `all_snakes` that prints all possible snakes with length MAX of a double-N game.

Example: for a double-1 game all possibilities of length 3 are:

```
[0:0] [0:1] [1:1]
[1:1] [1:0] [0:0]
```

Example: for a double-2 game all possibilities of length 6 are:

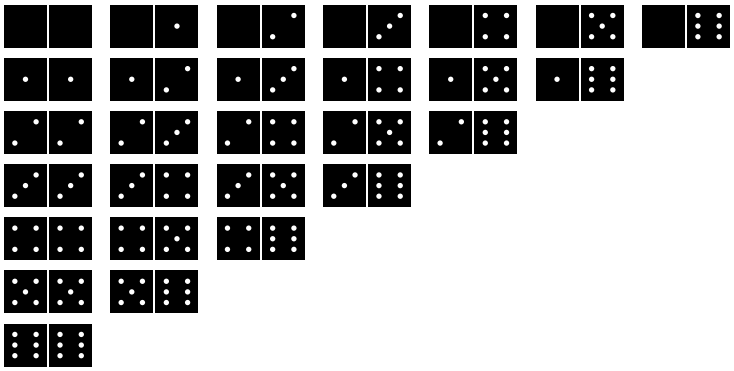

Figure 1: The dubble-six set of dominoes.

```
[(0,2),(2,2),(2,1),(1,1),(1,0),(0,0)]
[(0,1),(1,1),(1,2),(2,2),(2,0),(0,0)]
[(1,1),(1,2),(2,2),(2,0),(0,0),(0,1)]
[(0,0),(0,2),(2,2),(2,1),(1,1),(1,0)]
[(2,2),(2,1),(1,1),(1,0),(0,0),(0,2)]
[(0,0),(0,1),(1,1),(1,2),(2,2),(2,0)]
[(1,2),(2,2),(2,0),(0,0),(0,1),(1,1)]
[(1,0),(0,0),(0,2),(2,2),(2,1),(1,1)]
[(2,2),(2,0),(0,0),(0,1),(1,1),(1,2)]
[(1,1),(1,0),(0,0),(0,2),(2,2),(2,1)]
[(2,1),(1,1),(1,0),(0,0),(0,2),(2,2)]
[(2,0),(0,0),(0,1),(1,1),(1,2),(2,2)]
```

Exercise 3.37 (Bag packer). In many so-called *role playing games* the player controls an avatar that through adventures (*quests*) can collect objects. A typical assortment consists of gems (light, small, worth a lot), potions (small, somewhat heavier, not very expensive), weaponry (swords, axes, bows, etc.; big, heavy, and varying in price from cheap to very expensive), armour (helmets, chainmail, gloves, boots, shoulder pads, etc.; all varying in size, weight and value), books (small, somewhat heavier, sometimes expensive) and what you find in these kinds of worlds (enchanted equipment such as rings, weapons, armour). Some objects are *stackable*. These objects have certain dimensions and weight, but multiple objects can take up the same space at the same time. This does lead to an increase of total and weight, but not in the taken up space. Typical *stackable* objects are gold, gems, potions, rings, etc.

The whole of items that a player collects is called their *inventory*. This inventory always has a certain *limited* capacity. This capacity is sometimes only decided by numbers, weight or dimensions, and often by a combination of these factors.

In the game *Skyrim* objects have a weight and a value. All objects are *stackable*. The inventory is only limited by the common weight: it may not be heavier than a predetermined value.

The lead character wants to fill his inventory as optimally as possible with the items he obtained in-game (whether legitimately or illegitimately). The player is not attached to any object and plans to sell them to replenish his bag of money. This means the player wants to fill his inventory in such a way that it has the maximum possible value; there should be no other com-

bination of items of which the sum of values is greater (although it may be the same). For our protagonist, write a function that takes the following arguments:

1. A capacity of the maximal weight.
2. A list of objects that each have a value and weight.

The function returns a collection of items that has the maximum summed value. Design appropriate data structures for this assignment.

Exercise 3.38 (Bag packer, part II). In exercise 3.37 you have designed a function that filles a bag pack with items to get the maximum possible value. The criterion to stop stuffing was the maximum weight that the bag pack could carry. In this exercise we introduce the *dimensions* criterium, via the computer game *Neverwinter Nights*. In this game objects have not just a weight and a value, but also *dimensions* (width and height). The inventory is carried in a certain backpack, which also has a certain width and height. In the game the inventory is limited by the dimensions: objects may not overlap. Some items in *Neverwinter Nights* are *stackable*, some are not. Typical examples of *stackable* objects are potions, arrows, rings, etc. Typical non-*stackable* objects are swords, armour, helmets, and so on.

A special element of *Neverwinter Nights* is the so-called *bag of holding*: this is a bag pack you can place in your inventory, and in which you can then place further items. Every *bag of holding* has its own dimensions, value and capacity (which is usually much bigger than its dimensions — e.g. a *bag of holding* with dimensions 2×2 units may have an internal capacity of 30×25 units). Obviously, these items are treasured by adventurers because these significantly increase the capacity of a player's inventory.

Again, write a function for the adventurer to get an optimal contents for their bag pack, such that the sale of the items gains the maximal amount of profit. Write a function that takes the following arguments:

1. A bag pack with a certain width and height;
2. A list of objects that each have a width and height, a value, and a weight. Each object belongs to a kind, and the kind may or may not be *stackable*.

The function needs to return the contents of the bag pack for which the sum of the values of all objects is maximal, i.e. there is no other combination of items that fit in the bag pack for which the sum of the values is *greater* (although it may be equal).

Exercise 3.39 (Farmer wants a wife). In the "stable marriage" problem, a set of N suitors needs to be coupled with N partners such that these result in 'stable' relationships. Each person has supplied an ordered list of preferences for all candidates on the other side, through a numbering of 1 to N, where 1 indicates the most preferred candidate and N the least preferred candidate. A coupling of all suitors and partners is stable when there is no suitor and partner that would rather be with each other than with their current 'better half'.

If N is an even number, then there is an algorithm that computes such a stable coupling. This algorithm was proven correct in 1962 by David Gale and Lloyd Shapely[7]. It is an iterative

[7] Source: `http://en.wikipedia.org/wiki/Stable_marriage_problem`.

algorithm that keeps computing the following: find a not yet coupled suitor and find the most favoured partner to whom the suitor has not proposed yet. The suitor is rejected if the proposed partner is already coupled with a suitor that is higher on the proposed partner's list of favourites. In every other case they accept. This means that if the proposed partner was already matched up with a suitor (which should then be lower on her list of favourites), then that suitor is no longer matched. This repeats until all suitors are coupled.

Implement this algorithm. Do this by implementing the following function:

```
type  Nr = Int     -- 1..N
farmer_wants_a_wife :: ([[Nr]],[[Nr]]) → [(Nr,Nr)]
```

The expression `(farmer_wants_a_wife (preferences_suitors, preferences_partners))` computes a 'stable marriage' solution between the population of suitors and partners using the Gale / Shapely algorithm, if the input corresponds to the following properties:

1. The length `N` of `preferences_suitors` is identical to the length of `preferences_partners` and is even.

2. The preferences of each suitor and partner are a permutation of `[1 .. N]`.

The solution is a list of couples (suitor, partner) that is stable.

Exercise 3.40 (A plan for the improvement of English spelling).

> *For example, in Year 1 that useless letter* c *would be dropped to be replased either by* k *or* s*, and likewise* x *would no longer be part of the alphabet. The only kase in which* c *would be retained would be the* ch *formation, which will be dealt with later.*
>
> *Year 2 might reform* w *spelling, so that* which *and* one *would take the same konsonant, wile Year 3 might well abolish* y *replasing it with* i *and Iear 4 might fiks the* g/j *anomali wonse and for all.*
>
> *Jenerally, then, the improvement would kontinue iear bai iear with Iear 5 doing awai with useless double konsonants, and Iears 6-12 or so modifaiing vowlz and the rimeining voist and unvoist konsonants.*
>
> *Bai Iear 15 or sou, it wud fainali bi posibl tu meik ius ov thi ridandant letez* c, y *and* x *– bai now jast a memori in the maindz ov ould doderez – tu riplais* ch, sh*, and* th *rispektivli.*
>
> *Fainali, xen, aafte sam 20 iers ov orxogrefkl riform, wi wud hev a lojikl, kohirnt speling in ius xrewawt xe Ingliy-spiking werld.* (Mark Twain)

Write a program that converts an English text (e.g. `ImproveEnglishSpelling.txt` in the exercise folder) by the rules described above. The text leaves some room for interpretation of the new spelling rules. Instead, implement the following approximation of Twain's proposal for improvement of the English Spelling. These rules must be applied *consecutively* (make sure to account for capital letters):

1. 'ch' remains unchanged; 'c' followed by 'e' or 'i' becomes 's'; 'c' becomes 'k' in every other case.

2. 'x' becomes 'ks'

3. ‘wh’ becomes ‘w’
4. ‘y’ becomes ‘i’
5. ‘g’ followed by ‘e’, ‘i’, ‘y’ becomes ‘j’ except for *gear*, *get*, *give*, *gir*, *gift*.
6. All double consonants are reduced to single consonants.
7. ‘ph’ becomes ‘f’
8. ‘r’ is removed if it is at the end of a word and the next word starts with a consonant.
9. ‘ch’ is replaced by ‘c’; ‘sh’ is replaced by ‘y’; ‘th’ is replaced by ‘x’.

Hints to practitioners 3. Do not confuse `[Base]` with the singleton list `[base]`. The first is a type; the second is a value, a shorthand for `base :[]`. Identifiers for values and identifiers for types live in different name spaces. Hence it is actually possible to use the same name for a type variable and for a value variable e.g.

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

The occurrence of `a` on the first line is a type variable; the occurrence of `a` on the second line is a value variable of type `a :: a`. If you think that this is confusing, simply use different names e.g.

```
insert :: (Ord elem) ⇒ elem → [elem] → [elem]
```

However, keep this “feature” in the back of your head, if you read other people’s code. There is a, perhaps, unfortunate tendency to use the same names both for types and elements of types.