Exercise 7.2

Give a pattern-matching definition of a function which:

1. adds together the first two integers in a list, if a list contains at least two elements;

2. returns the head element if the list contains one

3. returns zero otherwise.

Exercise 7.2

Give a pattern-matching definition of a function which:

1. adds together the first two integers in a list, if a list contains at least two elements;

2. returns the head element if the list contains one

3. returns zero otherwise.

```
sumFirstTwo :: [Integer] -> Integer
sumFirstTwo (x:_)
```

Exercise 7.2

Give a pattern-matching definition of a function which:

1. adds together the first two integers in a list, if a list contains at least two elements;

2. returns the head element if the list contains one

3. returns zero otherwise.

```
sumFirstTwo :: [Integer] -> Integer
sumFirstTwo (x:y:_) = x + y
```

Exercise 7.2

Give a pattern-matching definition of a function which:

1. adds together the first two integers in a list, if a list contains at least two elements;

2. returns the head element if the list contains one

3. returns zero otherwise.

```
sumFirstTwo :: [Integer] -> Integer
sumFirstTwo (x:y:_) = x + y
sumFirstTwo (x:_)   = x
```

Exercise 7.2

Give a pattern-matching definition of a function which:

1. adds together the first two integers in a list, if a list contains at least two elements;

2. returns the head element if the list contains one

3. returns zero otherwise.

```
sumFirstTwo :: [Integer] -> Integer
sumFirstTwo (x:y:_) = x + y
sumFirstTwo (x:_)   = x
sumFirstTwo []      = 0
```

Exercise 7.6

Define the functions ‘and2’ and ‘or2’ on a list of Booleans
These should return the conjuction or disjunction on a list of booleans
Note that these are already defined in the prelude, so call them and2 and or2

Example:

```
and2 [True, False] = False
or2 [False, True] = TRUE
--------------------------------------------------------------
```

Exercise 7.6

Define the functions ‘and2’ and ‘or2’ on a list of Booleans
These should return the conjuction or disjunction on a list of booleans
Note that these are already defined in the prelude, so call them and2 and or2

Example:

```
and2 [True, False] = False
or2 [False, True] = TRUE
-----------------------------------------------------------
and2 :: [Bool] -> Bool
and2 [] = True
```

Exercise 7.6

Define the functions ‘and2’ and ‘or2’ on a list of Booleans
These should return the conjuction or disjunction on a list of booleans
Note that these are already defined in the prelude, so call them and2 and or2

Example:

```
and2 [True, False] = False
or2 [False, True] = TRUE
----------------------------------------------------------
and2 :: [Bool] -> Bool
and2 [] = True
and2 (b:bs) = b && (and2 bs)
```

Exercise 7.6

Define the functions ‘and2’ and ‘or2’ on a list of Booleans
These should return the conjuction or disjunction on a list of booleans
Note that these are already defined in the prelude, so call them and2 and or2

Example:

```
and2 [True, False] = False
or2 [False, True] = TRUE
-----------------------------------------------------------
and2 :: [Bool] -> Bool
and2 [] = True
and2 (b:bs) = b && (and2 bs)

or2 :: [Bool] -> Bool
or2 [] = False
or2 (b:bs) = b || (or2 bs)
```

Exercise 7.8

Using primitive recursion over lists, define a function:

```
elemNum :: Integer -> [Integer] -> Integer
```

so that `elemNum x xs` returns the number of times that `x` occurs in the list `xs`.

Can you define `elemNum` without primitive recursion, using list comprehensions and built-in functions instead?

Exercise 7.8

Using primitive recursion over lists, define a function:

```
elemNum :: Integer -> [Integer] -> Integer
```

so that `elemNum x xs` returns the number of times that `x` occurs in the list `xs`.

Can you define `elemNum` without primitive recursion, using list comprehensions and built-in functions instead?

```
elemNum :: Integer -> [Integer] -> Integer
elemNum a [] = 0
```

Exercise 7.8

Using primitive recursion over lists, define a function:

`elemNum :: Integer -> [Integer] -> Integer`

so that `elemNum x xs` returns the number of times that `x` occurs in the list `xs`.

Can you define `elemNum` without primitive recursion, using list comprehensions and built-in functions instead?

```
elemNum :: Integer -> [Integer] -> Integer
elemNum a [] = 0
elemNum a (x:xs)
  | a == x     = 1 + elemNum a xs
```

Exercise 7.8

Using primitive recursion over lists, define a function:

```
elemNum :: Integer -> [Integer] -> Integer
```

so that `elemNum x xs` returns the number of times that `x` occurs in the list `xs`.

Can you define `elemNum` without primitive recursion, using list comprehensions and built-in functions instead?

```
elemNum :: Integer -> [Integer] -> Integer
elemNum a [] = 0
elemNum a (x:xs)
   | a == x       = 1 + elemNum a xs
   | otherwise    = elemNum a xs
```

Exercise 7.8

Using primitive recursion over lists, define a function:

```
elemNum :: Integer -> [Integer] -> Integer
```

so that `elemNum x xs` returns the number of times that `x` occurs in the list `xs`.

Can you define `elemNum` without primitive recursion, using list comprehensions and built-in functions instead?

```
elemNum :: Integer -> [Integer] -> Integer
elemNum a [] = 0
elemNum a (x:xs)
   | a == x      = 1 + elemNum a xs
   | otherwise   = elemNum a xs

elemNum2 :: Integer -> [Integer] -> Integer
elemNum2 a xs = sum [  | x<-xs,       ]
```

Exercise 7.8

Using primitive recursion over lists, define a function:

```
elemNum :: Integer -> [Integer] -> Integer
```

so that `elemNum x xs` returns the number of times that `x` occurs in the list `xs`.

Can you define `elemNum` without primitive recursion, using list comprehensions and built-in functions instead?

```
elemNum :: Integer -> [Integer] -> Integer
elemNum a [] = 0
elemNum a (x:xs)
   | a == x      = 1 + elemNum a xs
   | otherwise  = elemNum a xs

elemNum2 :: Integer -> [Integer] -> Integer
elemNum2 a xs = sum [1 | x<-xs, x==a  ]
```

Exercise 7.9

- Define a function :

  `unique :: [Integer] -> [Integer]`

  so that `unique xs` returns the list of elements of `xs` which occurs exactly once.

  Example:

  `unique [4, 2, 1, 3, 2, 3] = [4, 1]`

- You might like to think of two solutions to this problem: one using list comprehensions and the other not.

Exercise 7.9

```
unique :: [Integer] -> [Integer]
unique xs = [x | x<-xs, elemNum x xs == 1]
```

Exercise 7.9

```
unique :: [Integer] -> [Integer]
unique xs = [x | x<-xs, elemNum x xs == 1]

removeElement :: Integer -> [Integer] -> (Bool,[Integer])
removeElement _ [] = (False, [])
```

Exercise 7.9

```
unique :: [Integer] -> [Integer]
unique xs = [x | x<-xs, elemNum x xs == 1]

removeElement :: Integer -> [Integer] -> (Bool,[Integer])
removeElement _ [] = (False, [])
removeElement x (y:ys)
  | x == y     = (True, rem)
  |
 where (flag,rem) = removeElement x ys
```

Exercise 7.9

```
unique :: [Integer] -> [Integer]
unique xs = [x | x<-xs, elemNum x xs == 1]

removeElement :: Integer -> [Integer] -> (Bool,[Integer])
removeElement _ [] = (False, [])
removeElement x (y:ys)
  | x == y      = (True, rem)
  | otherwise   = (flag, y:rem)
  where (flag,rem) = removeElement x ys
```

Exercise 7.9

```
unique :: [Integer] -> [Integer]
unique xs = [x | x<-xs, elemNum x xs == 1]

removeElement :: Integer -> [Integer] -> (Bool,[Integer])
removeElement _ [] = (False, [])
removeElement x (y:ys)
  | x == y     = (True, rem)
  | otherwise  = (flag, y:rem)
  where (flag,rem) = removeElement x ys

unique2 :: [Integer] -> [Integer]
unique2 [] = []
```

Exercise 7.9

```
unique :: [Integer] -> [Integer]
unique xs = [x | x<-xs, elemNum x xs == 1]

removeElement :: Integer -> [Integer] -> (Bool,[Integer])
removeElement _ [] = (False, [])
removeElement x (y:ys)
  | x == y      = (True, rem)
  | otherwise   = (flag, y:rem)
  where (flag,rem) = removeElement x ys

unique2 :: [Integer] -> [Integer]
unique2 [] = []
unique2 (x:xs)
  | flag      = unique2 rem
  |
```

Exercise 7.9

```
unique :: [Integer] -> [Integer]
unique xs = [x | x<-xs, elemNum x xs == 1]

removeElement :: Integer -> [Integer] -> (Bool,[Integer])
removeElement _ [] = (False, [])
removeElement x (y:ys)
  | x == y      = (True, rem)
  | otherwise   = (flag, y:rem)
  where (flag,rem) = removeElement x ys

unique2 :: [Integer] -> [Integer]
unique2 [] = []
unique2 (x:xs)
  | flag       = unique2 rem
  |
  where (flag,rem) = removeElement x xs
```

Exercise 7.9

```
unique :: [Integer] -> [Integer]
unique xs = [x | x<-xs, elemNum x xs == 1]

removeElement :: Integer -> [Integer] -> (Bool,[Integer])
removeElement _ [] = (False, [])
removeElement x (y:ys)
  | x == y      = (True, rem)
  | otherwise   = (flag, y:rem)
  where (flag,rem) = removeElement x ys

unique2 :: [Integer] -> [Integer]
unique2 [] = []
unique2 (x:xs)
  | flag       = unique2 rem
  | otherwise = x : (unique2 rem)
  where (flag,rem) = removeElement x xs
```

Exercise 7.16

- By modifying the definition of the `ins` function we can change the behaviour of the sort, `iSort.` Redefine `ins` in two different ways so that

  1 : the list is sorted in descending order;

  2: duplicates are removed from the list.

Exercise 7.16

- By modifying the definition of the `ins` function we can change the behaviour of the sort, `iSort.` Redefine `ins` in two different ways so that

  1 : the list is sorted in descending order;

  2: duplicates are removed from the list.

```
ins :: Integer -> [Integer] -> [Integer]

ins x [] = [x]
```

Exercise 7.16

- By modifying the definition of the `ins` function we can change the behaviour of the sort, `iSort.` Redefine `ins` in two different ways so that

  1 : the list is sorted in descending order;

  2: duplicates are removed from the list.

```
ins :: Integer -> [Integer] -> [Integer]
ins x [] = [x]
ins x (y:ys)
  | x == y      = y:ys
```

Exercise 7.16

- By modifying the definition of the `ins` function we can change the behaviour of the sort, `iSort.` Redefine `ins` in two different ways so that

  1 : the list is sorted in descending order;

  2: duplicates are removed from the list.

```
ins :: Integer -> [Integer] -> [Integer]
ins x [] = [x]
ins x (y:ys)
   | x == y      = y:ys
   | x > y       = x:(y:ys)
```

Exercise 7.16

- By modifying the definition of the `ins` function we can change the behaviour of the sort, `iSort`. Redefine `ins` in two different ways so that

  1 : the list is sorted in descending order;

  2: duplicates are removed from the list.

```
ins :: Integer -> [Integer] -> [Integer]

ins x [] = [x]

ins x (y:ys)

   | x == y      = y:ys

   | x > y       = x:(y:ys)

   | otherwise   = y : ins x ys
```

Exercise 7.16

- By modifying the definition of the `ins` function we can change the behaviour of the sort, `iSort`. Redefine `ins` in two different ways so that

  1 : the list is sorted in descending order;

  2: duplicates are removed from the list.

```
ins :: Integer -> [Integer] -> [Integer]
ins x [] = [x]
ins x (y:ys)
   | x == y     = y:ys
   | x > y      = x:(y:ys)
   | otherwise  = y : ins x ys

iSort :: [Integer] -> [Integer]
iSort [] = []
iSort (x:xs) = ins x _
```

Exercise 7.16

- By modifying the definition of the `ins` function we can change the behaviour of the sort, `iSort`. Redefine `ins` in two different ways so that

  1 : the list is sorted in descending order;

  2: duplicates are removed from the list.

```
ins :: Integer -> [Integer] -> [Integer]
ins x [] = [x]
ins x (y:ys)
   | x == y      = y:ys
   | x > y       = x:(y:ys)
   | otherwise   = y : ins x ys

iSort :: [Integer] -> [Integer]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

Exercise 7.25

- One list is a sublist of another if the elements of the first occur in the second, in the same order. For instance, "ship" is a sublist of "Fish & Chips", but not of "hippies".

  A list is a subsequence of another if it occurs as a sequence of elements next to each other.

- For example, "Chip" is a subsequence of "Fish & Chips", but not of "Chin up".
- Define functions which decide whether one string is a sublist or a subsequence of another string.

Exercise 7.25

```
isSubStr :: String -> String -> Bool
isSubStr [] _ = True
```

Exercise 7.25

```
isSubStr :: String -> String -> Bool
isSubStr [] _ = True
isSubStr _ [] = False
```

Exercise 7.25

```
isSubStr :: String -> String -> Bool
isSubStr [] _ = True
isSubStr _ [] = False
isSubStr (x:xs) (y:ys)
  | x == y    = isSubStr xs ys
  |
```

Exercise 7.25

```
isSubStr :: String -> String -> Bool
isSubStr [] _ = True
isSubStr _ [] = False
isSubStr (x:xs) (y:ys)
  | x == y     = isSubStr xs ys
  | otherwise = isSubStr (x:xs) ys
```

Exercise 7.25

```
isSubStr :: String -> String -> Bool
isSubStr [] _ = True
isSubStr _ [] = False
isSubStr (x:xs) (y:ys)
  | x == y    = isSubStr xs ys
  | otherwise = isSubStr (x:xs) ys

isSubSeq :: String -> String -> Bool
isSubSeq [] _ = True
isSubSeq _ [] = False
```

Exercise 7.25

```
isSubStr :: String -> String -> Bool
isSubStr [] _ = True
isSubStr _ [] = False
isSubStr (x:xs) (y:ys)
  | x == y    = isSubStr xs ys
  | otherwise = isSubStr (x:xs) ys

isSubSeq :: String -> String -> Bool
isSubSeq [] _ = True
isSubSeq _ [] = False
isSubSeq (x:xs) (y:ys)
  | x == y    = (unzip(zip (x:xs) (y:ys))) == ((x:xs)
(x:xs)) || isSubSeq (x:xs) ys
  |
```

Exercise 7.25

```
isSubStr :: String -> String -> Bool
isSubStr [] _ = True
isSubStr _ [] = False
isSubStr (x:xs) (y:ys)
   | x == y    = isSubStr xs ys
   | otherwise = isSubStr (x:xs) ys

isSubSeq :: String -> String -> Bool
isSubSeq [] _ = True
isSubSeq _ [] = False
isSubSeq (x:xs) (y:ys)
   | x == y    = (unzip(zip (x:xs) (y:ys))) == ((x:xs)
(x:xs)) || isSubSeq (x:xs) ys
   | otherwise = isSubSeq (x:xs) ys
```

Exercise 7.33

Define a function `isPalin` which tests whether a string is a palindrome.

Example of a palindrome: “Madam I’m Adam”

Note that punctuation and white space are ignored, and that there is no distinction between capital and small letters.

Exercise 7.33

Define a function `isPalin` which tests whether a string is a palindrome.

Example of a palindrome: “Madam I’m Adam”

Note that punctuation and white space are ignored, and that there is no distinction between capital and small letters.

```
isPalin :: String -> Bool

isPalin str = s == reverse s

   where s = [toLower c | c <- str, isAlpha c]
```

Exercise 7.34

Design a function:

```
subst :: String -> String -> String -> String
```

so that:

```
subst oldSub newSub st
```

is the result of replacing the first occurence in `st` of the substring `oldSub` by the substring `newSub`. For instance:

```
subst "much " "tall " "How much is that?" = "How tall is that?"
---------------------------------------------------------------
```

Exercise 7.34

Design a function:

```
subst :: String -> String -> String -> String
```

so that:

```
subst oldSub newSub st
```

is the result of replacing the first occurence in `st` of the substring `oldSub` by the substring `newSub`. For instance:

```
subst "much " "tall " "How much is that?" = "How tall is that?"
-----------------------------------------------------------
subst :: String -> String -> String -> String
subst _ _ [] = []
```

Exercise 7.34

Design a function:

```
subst :: String -> String -> String -> String
```

so that:

```
subst oldSub newSub st
```

is the result of replacing the first occurence in `st` of the substring `oldSub` by the substring `newSub`. For instance:

```
subst "much " "tall " "How much is that?" = "How tall is that?"
-------------------------------------------------------------
subst :: String -> String -> String -> String
subst _ _ [] = []
subst (x:xs) sub (y:ys)
  | match     = sub ++ (drop (length xs) ys)
  |
 where match = (unzip(zip (x:xs) (y:ys))) == ((x:xs),(x:xs))
```

Exercise 7.34

Design a function:

```
subst :: String -> String -> String -> String
```

so that:

```
subst oldSub newSub st
```

is the result of replacing the first occurence in `st` of the substring `oldSub` by the substring `newSub`. For instance:

```
subst "much " "tall " "How much is that?" = "How tall is that?"
-------------------------------------------------------------------
subst :: String -> String -> String -> String
subst _ _ [] = []
subst (x:xs) sub (y:ys)
  | match     = sub ++ (drop (length xs) ys)
  | otherwise = y:(subst (x:xs) sub ys)
 where match = (unzip(zip (x:xs) (y:ys))) == ((x:xs),(x:xs))
```