Tutoraufgabe 1 (Überblickswissen):

a) In Haskell sind Funktionen gleichberechtigte Datenobjekte. Was bedeutet das? Welche Vorteile bietet es?

b) Typdeklarationen sind in Haskell nicht notwendig. Welche Gründe gibt es, trotzdem eine anzugeben?

c) In der Mathematik sind wir es gewohnt, eine Funktion mit ihrem Definitions- und Zielbereich anzugeben, zum Beispiel $sqrt : \mathbb{N} \rightarrow \mathbb{R}$. Die Typdeklaration einer Funktion in Haskell dagegen kann mehrere solcher Pfeile enthalten, zum Beispiel `isSquare :: Int -> Int -> Bool`. Was bedeutet das und welche Vorteile bietet es gegenüber der Typdeklaration `isSquare :: (Int,Int) -> Bool`?

Lösung:

a) Eine Funktion kann in Haskell nicht nur mit bestimmten Parametern ausgewertet werden, sie kann auch selbst Parameter sein. Eine andere Funktion bekommt also als Parameter eine Funktion übergeben, wodurch diese übergebene Funktion in der Auswertung der anderen Funktion genutzt werden kann. Außerdem kann eine Funktion auch eine Funktion zurückgeben. Solche Funktionen, die mit anderen Funktionen umgehen, nennt man *Funktionen höherer Ordnung*. Dieses Konzept erlaubt eine hohe Flexibilität in der Implementierung, da wir das Verhalten an bestimmten Stellen über die übergebenen Funktionen je nach Situation anpassen können. So können wir Funktionen höherer Ordnung schreiben und nutzen, die nicht nur für einen, sondern für viele Zwecke taugen.

b) Die Angabe einer Typdeklaration ist aus mehreren Gründen sinnvoll:

- Die Typen der Parameter und des Rückgabewerts geben zwar noch nicht an, was genau die Funktion tut, aber immerhin, womit sie arbeitet. Diese Übersicht ist bereits sehr hilfreich, wenn man den Code später selbst erneut liest. Durch die Typdeklaration versteht man schneller, was hier implementiert wurde.
- Gleiches gilt natürlich auch für andere Entwickler*innen, die sich den Code ansehen. Außerdem dient die Typdeklaration für sie wie eine Art Schnittstelle, denn in den meisten Fällen ist für andere bei der Nutzung nur interessant, wieviele und welche Typen die Funktion erwartet und welchen Typ die Funktion zurückgibt. Die genaue Implementierung interessiert hier, wie auch in anderen Programmiersprachen, meist nicht.
- Beim Programmieren ist es hilfreich, schon vor der Implementierung genaue Vorstellungen davon zu haben, was man eigentlich tun möchte. Die Typdeklaration mit ihren wichtigen Rahmendaten hilft dabei, da sie Kenntnis über die wichtigsten Typen erzwingt.
- Die Angabe der Typdeklaration erhöht die Wahrscheinlichkeit, Programmierfehler zu finden. Wenn die Implementierung einer Funktion nicht zu dem beabsichtigten (deklarierten) Typ passt, dann wird dies bei der automatischen Typüberprüfung bemerkt.

c) Eine Typdeklaration mit mehreren Pfeilen kann zunächst so verstanden werden, dass mehrere Parameter erwartet werden, damit ein konkreter Wert zurückgegeben wird. So wird im Falle von `isSquare` entweder `True` oder `False` zurückgegeben, wenn man der Funktion zwei Werte vom Typ `Int` als Parameter gibt. Wenn man die Typdeklaration nur so liest, mutet die Notation vielleicht noch etwas seltsam an. Man kann die Funktion aber auch anders benutzen, nämlich nur mit einem statt mit zwei Parametern. Dann gibt die Funktion keinen Wert zurück, sondern wieder eine Funktion. Diese zurückgegebene Funktion gibt dann, wenn sie auf dem ursprünglich zweiten Parameter ausgewertet wird, den erwarteten Rückgabewert zurück. Der erste Pfeil in der ursprünglichen Typdeklaration bedeutet also: Diese Funktion bildet von einem Wert vom Typ `Int` auf eine andere Funktion ab, die von einem Wert vom Typ `Int` auf einen Wert vom Typ `Bool` abbildet. Der zweite Pfeil in der Typdeklaration ist dann sozusagen der Pfeil dieser zurückgegebenen Funktion. Wenn wir der ursprünglichen Funktion direkt zwei Parameter übergeben, merken wir von dieser zurückgegebenen Funktion gar nichts, da sie direkt auf dem zweiten Parameter ausgewertet wird.

Arbeiten wir mit der Variante der Funktion, die ein Tupel aus zwei `Int`-Werten erwartet, ergibt sich dieses gerade beschriebene Verhalten nicht, sondern es werden wirklich zwei Werte auf einmal erwartet. Sie werden zusammen wie ein Parameter behandelt. Die Umwandlung von der Variante mit Tupel in die Form ohne Tupel nennt man *Currying*, die Rückumwandlung nennt man *Uncurrying*.

Tutoraufgabe 2 (Auswertungsstrategie):

Gegeben sei das folgende Haskell-Programm:

```
second :: [Int] -> Int
second []        = 0
second (_:[])    = 0
second (_:x:xs)  = x

doubleEach :: [Int] -> [Int]
doubleEach []     = []
doubleEach (x:xs) = x * 2 : (doubleEach xs)

repeat :: Int -> Int -> [Int]
repeat x n = if n > 0 then x : (repeat x (n-1)) else []
```

Die Funktion `second` gibt zu jeder Eingabeliste mit mindestens zwei Elementen den zweiten Eintrag der Liste zurück. Für Listen mit weniger als zwei Elementen wird `0` zurückgegeben. Die Funktion `doubleEach` gibt die Liste zurück, die durch Verdoppeln jedes Elements aus der Eingabeliste entsteht. Der Aufruf `doubleEach [1, 2, 3, 1]` würde also `[2, 4, 6, 2]` ergeben. Die Funktion `repeat` erzeugt eine Liste, die das erste Argument so oft enthält, wie das zweite Argument angibt. Beispielsweise erhält man beim Aufruf `repeat 5 3` die Liste `[5, 5, 5]` als Rückgabe.
Geben Sie alle Zwischenschritte bei der Auswertung des Ausdrucks

`repeat (second (doubleEach [2, 3, 5])) (second [3, 1, 4])`

an. Unterstreichen Sie vor jedem Auswertungsschritt den Teil des Ausdrucks, der als Nächstes an seiner äußersten Position ausgewertet wird. Schreiben Sie hierbei (um Platz zu sparen) `s`, `d` und `r` statt `second`, `doubleEach` und `repeat`.

Hinweise:

- Beachten Sie, dass Haskell eine Leftmost-Outermost Lazy Auswertungsstrategie besitzt. Allerdings sind Operatoren wie `*`, `>` und `-`, die auf eingebauten Zahlen arbeiten, strikt, d. h. hier müssen vor Anwendung des Operators seine Argumente vollständig ausgewertet worden sein (wobei zunächst das linke Argument ausgewertet wird).
- `_` bezeichnet den sogenannten Joker-Pattern, der für einen beliebigen Wert steht. Statt des Joker-Patterns könnte man also auch eine neue Variable benutzen. Die zweite definierende Gleichung von `second` könnte also auch wie folgt lauten: `second (y:[]) = 0`.

Lösung:

```
r (s (d [2, 3, 5])) (s [3, 1, 4])
→ if s [3, 1, 4] > 0 then s (d [2, 3, 5]) : r (s (d [2, 3, 5])) (s [3, 1, 4] - 1) else []
→ if 1 > 0 then s (d [2, 3, 5]) : r (s (d [2, 3, 5])) (1 - 1) else []
→ if True then s (d [2, 3, 5]) : r (s (d [2, 3, 5])) (1 - 1) else []
→ s (d [2, 3, 5]) : r (s (d [2, 3, 5])) (1-1)
→ s (2*2 : d [3, 5]) : r (s (2*2 : d [3, 5])) (1-1)
→ s (2*2 : 3*2 : d [5]) : r (s (2*2 : 3*2 : d [5])) (1-1)
→ 3*2 : r (3*2) (1-1)
→ 6 : r 6 (1-1)
→ 6 : if (1 - 1) > 0 then 6 : r 6 ((1-1) - 1) else []
→ 6 : if 0 > 0 then 6 : r 6 (0 - 1) else []
→ 6 : if False then 6 : r 6 (0 - 1) else []
→ 6 : [] = [6]
```

Aufgabe 3 (Auswertungsstrategie): (26 Punkte)

Gegeben sei das folgende Haskell-Programm:

```
doubleElements :: [Int] -> [Int]
doubleElements [] = []
doubleElements (x:xs) = x : x : doubleElements xs

firstN :: Int -> [Int] -> [Int]
firstN n [] = []
firstN n (x:xs) =
  if n > 0 then x : firstN (n-1) xs else []

listSum :: [Int] -> Int
listSum [] = 0
listSum (x:xs) = x + listSum xs
```

Die Funktion `doubleElements` berechnet eine Liste, welche aus einer gegebenen Liste hervorgeht, indem jedes Element dieser Liste noch einmal hinter sich selbst eingefügt wird. Zum Beispiel wertet der Ausdruck `doubleElements [1,2,3]` zu `[1,1,2,2,3,3]` aus. Die Funktion `firstN` gibt zu jeder Liste `xs` den längsten Anfang `ys` von `xs` zurück, sodass `ys` höchstens eine gegebene Anzahl an Elementen besitzt. Der Ausdruck `firstN 2 [1,2,3]` wertet beispielsweise zu `[1,2]` aus, während hingegen `firstN 5 [1,2,3]` zu `[1,2,3]` auswertet. Die Funktion `listSum` bildet die Summe über alle Elemente einer Liste vom Typ `[Int]`. Wird `listSum [1,2,3]` aufgerufen, so ergibt sich `6`.
Geben Sie alle Zwischenschritte bei der Auswertung des Ausdrucks

`listSum (firstN (1+0) (doubleElements [1+0, 1+0]))`

an. Unterstreichen Sie vor jedem Auswertungsschritt den Teil des Ausdrucks, der als Nächstes an seiner äußersten Position ausgewertet wird. Um Platz zu sparen können Sie hierbei `dE`, `fN` und `lS` statt `doubleElements`, `firstN` und `listSum` schreiben.

Hinweise:

- Beachten Sie, dass Haskell eine Leftmost-Outermost Lazy Auswertungsstrategie besitzt. Allerdings sind Operatoren wie `*`, `>` und `-`, die auf eingebauten Zahlen arbeiten, strikt, d. h. hier müssen vor Anwendung des Operators seine Argumente vollständig ausgewertet worden sein (wobei zunächst das linke Argument ausgewertet wird).

Lösung:

```
  listSum (firstN (1+0) (doubleElements [1+0,1+0]))
→ listSum (firstN (1+0) ((1+0):(1+0):doubleElements [1+0]))
→ listSum (if 1+0 > 0 then (1+0):firstN ((1+0) - 1) ((1+0):doubleElements [1+0]) else [])
→ listSum (if 1 > 0 then (1+0):firstN (1 - 1) ((1+0):doubleElements [1+0]) else [])
→ listSum (if True then (1+0):firstN (1 - 1) ((1+0):doubleElements [1+0]) else [])
→ listSum ((1+0):firstN (1 - 1) ((1+0):doubleElements [1+0]))
→ (1+0) + listSum (firstN (1-1) ((1+0):doubleElements [1+0]))
→ 1 + listSum (firstN (1-1) (1:doubleElements [1+0]))
→ 1 + listSum (if 1-1 > 0 then 1:firstN (1-1-1) (doubleElements [1+0]) else [])
→ 1 + listSum (if 0 > 0 then 1:firstN (0-1) (doubleElements [1+0]) else [])
→ 1 + listSum (if False then 1:firstN (0-1) (doubleElements [1+0]) else [])
→ 1 + listSum []
→ 1 + 0
→ 1
```

Tutoraufgabe 4 (Listen in Haskell):

Seien `x`, `y` ganze Zahlen vom Typ `Int` und seien `xs` und `ys` Listen der Längen n und m vom Typ `[Int]`. Welche der folgenden Gleichungen zwischen Listen sind richtig und welche nicht? Begründen Sie Ihre Antwort. Falls es sich um syntaktisch korrekte Ausdrücke handelt, geben Sie für jede linke und rechte Seite auch an, wie viele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.
Beispiel: Die Liste `[[1,2,3],[4,5]]` hat den Typ `[[Int]]` und enthält 2 Elemente.

a) `x : ([y] ++ xs)` = `[x] ++ (y : xs)`

b) `x:[y]` = `x:y`

c) `x:ys:xs` = `(x:ys) ++ xs`

d) `[x,x,y] ++ (x:xs)` = `x:x:((y:[x]) ++ xs)`

e) `[]:[[[1]],[]]` = `[[],[1]]:[[]]`

Hinweise:

- Hierbei steht `++` für den Verkettungsoperator für Listen. Das Resultat von `xs ++ ys` ist die Liste, die entsteht, wenn die Elemente aus `ys` — in der Reihenfolge wie sie in `ys` stehen — an das Ende von `xs` angefügt werden.
  Beispiel: `[1,2] ++ [1,2,3]` = `[1,2,1,2,3]`
- Falls linke und rechte Seite gleich sind, genügt **eine** Angabe des Typs und der Elementzahl.

Lösung:

a) Beide Ausdrücke repräsentieren die gleiche Liste der Länge $n + 2$ und vom Typ `[Int]`.

b) Der rechte Ausdruck ist nicht typkorrekt, da die Variable `y` keine Liste ist. Sie kann somit nicht als zweites Argument des Konstruktors "`:`" verwendet werden. Der linke Ausdruck ist typkorrekt und repräsentiert eine Liste der Länge 2 vom Typ `[Int]`. Die Ausdrücke sind also nicht gleich.

c) Der linke Ausdruck ist nicht typkorrekt, da `x` und `ys` nicht den gleichen Typ haben, aber in der gleichen Liste enthalten sein sollen, und da `ys` nicht den gleichen Typ hat wie die Elemente von `xs`, aber ein Element von `xs` sein soll. Der rechte Ausdruck repräsentiert eine Liste der Länge $n + m + 1$ und ist vom Typ `[Int]`). Die Ausdrücke sind also nicht gleich.

d) Beides sind typkorrekte Listenausdrücke (äquivalent zu `[x,x,y,x] ++ xs` der Länge $n+4$ und ebenfalls vom Typ `[Int]`). Die Ausdrücke sind also gleich.

e) Der linke Ausdruck ergibt ausgeschrieben die Liste `[[],[[1]],[]]`, also eine dreielementige Liste, die Listen von Listen enthält. Der Typ der inneren Listen ist `[[Int]]`. Der Typ des gesamten Ausdrucks ist demnach `[[[Int]]]`. Der zweite Ausdruck ergibt ausgeschrieben die Liste `[[[],[1]],[]]`, also eine zweielementige Liste, die Listen von Listen enthält. Auch hier ist der Typ der inneren Listen `[[Int]]` und der Typ des gesamten Ausdrucks ist `[[[Int]]]`. Da die beiden Ausdrucke zwar den gleichen Typ haben, aber nicht die gleichen Listen darstellen, sind die Ausdrücke also nicht gleich.

Tutoraufgabe 5 (Listen in Haskell (Video)):

Seien `x`, `y`, `z` ganze Zahlen vom Typ `Int` und seien `xs` und `ys` Listen der Längen n und m vom Typ `[Int]`. Welche der folgenden Gleichungen zwischen Listen sind richtig und welche nicht? Begründen Sie Ihre Antwort. Falls es sich um syntaktisch korrekte Ausdrücke handelt, geben Sie für jede linke und rechte Seite auch an, wie viele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.
Beispiel: Die Liste `[[x,y],[z]]` hat den Typ `[[Int]]` und enthält 2 Elemente.

Hinweise:

- Hierbei steht `++` für den Verkettungsoperator für Listen. Das Resultat von `xs ++ ys` ist die Liste, die entsteht, wenn die Elemente aus `ys` — in der Reihenfolge wie sie in `ys` stehen — an das Ende von `xs` angefügt werden.
  Beispiel: `[1,2] ++ [1,2,3]` = `[1,2,1,2,3]`
- Falls linke und rechte Seite gleich sind, genügt **eine** Angabe des Typs und der Elementzahl.

a) `[] ++ [xs]` = `[] : [xs]`

b) `[[]] ++ [x]` = `[] : [x]`

c) `[x] ++ [y]` = `x : [y]`

d) `x:y:z:(xs ++ ys)` = `[x,y,z] ++ xs ++ ys`

e) `[(x:xs):[ys],[[]]]` = `(([]:[]):[]) ++ ([([x] ++ xs),ys]:[])`

Lösung:

a) Beide Ausdrücke haben den Typ `[[Int]]`. Jedoch hat die erste Liste ein Element, während die zweite Liste zwei Elemente besitzt. Die Ausdrücke sind also nicht gleich.

b) Beide Ausdrücke sind nicht typkorrekt. Daher würde die Gleichung in Haskell nicht gelten. Allerdings würden beide Ausdrücke die gleiche (ungültige) Liste darstellen, nämlich `[[],x]` (somit sind beide Antworten, ob die Gleichung gilt oder nicht, zulässig). Diese Liste ist nicht typkorrekt, da sie sowohl eine Liste als auch einen `Int` Wert enthält.

c) Die Gleichung gilt, denn beide Ausdrücke repräsentieren die Liste `[x,y]` vom Typ `[Int]`, welche zwei Elemente enthält.

d) Die Gleichung gilt, denn beide Ausdrücke repräsentieren die gleiche Liste, welche zuerst die Elemente `x`, `y`, `z`, anschließend die n Elemente der Liste `xs` und schließlich die m Elemente der Liste `ys` enthält. Diese Liste enthält also insgesamt $3+n+m$ Elemente und ist vom Typ `[Int]`.

e) Beide Ausdrücke sind vom gleichen Typ `[[[Int]]]` und sind Listen mit zwei Elementen. Diese Elemente sind auch noch gleich (einerseits die Liste `[[]]` und andererseits die Liste `[x:xs,ys]`), allerdings ist ihre Reihenfolge in den beiden Ausdrücken unterschiedlich, sodass die Gleichung nicht gilt.

Aufgabe 6 (Listen in Haskell): (4+5+5+5+5 = 24 Punkte)

Seien `x`, `y` und `z` ganze Zahlen vom Typ `Int` und seien `xs` und `ys` Listen der Längen n und m vom Typ `[Int]`. Welche der folgenden Gleichungen zwischen Listen sind richtig und welche nicht? Begründen Sie Ihre Antwort. Falls es sich um syntaktisch korrekte Ausdrücke handelt, geben Sie für jede linke und rechte Seite auch an, wie viele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.
Beispiel: Die Liste `[[x,y],ys]` hat den Typ `[[Int]]` und enthält 2 Elemente.

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

b) `(x:[y]):[xs]` = `[[x] ++ [y]] ++ [xs]`

c) `((x:[y]):[[z]++ys]):[]` = `x:y:z:ys`

d) `(x:ys):[] ++ [xs]` = `(x:ys++xs):[[]]`

e) `(x:y:([z]++ys)):[xs]` = `[x,y,z]:[ys] ++ [xs]`

Hinweise:

- Hierbei steht `++` für den Verkettungsoperator für Listen. Das Resultat von `xs ++ ys` ist die Liste, die entsteht, wenn die Elemente aus `ys` — in der Reihenfolge wie sie in `ys` stehen — an das Ende von `xs` angefügt werden.
  Beispiel: `[1,2] ++ [1,2,3]` = `[1,2,1,2,3]`
- Falls linke und rechte Seite gleich sind, genügt **eine** Angabe des Typs und der Elementzahl.

Lösung:

a) Der Teilausdruck `x:[ys]` des linken Ausdrucks ist nicht typkorrekt, da `[ys]` den Typ `[[Int]]` besitzt, jedoch wird hier ein Wert vom Typ `[Int]` erwartet, da `x::Int` gilt.

Der rechte Ausdruck ist eine Liste vom Typ `[[Int]]` und enthält 3 Elemente.

b) Beide Ausdrücke repräsentieren dieselbe Liste der Länge 2 vom Typ `[[Int]]`, nämlich `[[x,y],xs]`

c) Der linke Ausdruck entspricht der Liste `[[[x,y],[z]++ys]]` der Länge 1 vom Typ `[[[Int]]]`.

Der rechte Ausdruck repräsentiert eine Liste mit $3+m$ Elementen vom Typ `[Int]`, nämlich `[x,y,z]++ys`.

Beide Ausdrücke entsprechen also verschiedenen Listen.

d) Beide Ausdrücke sind vom Typ `[[Int]]` und besitzen die Länge 2. Jedoch enthält die rechte Liste im Gegensatz zur Linken den Wert `[]::[Int]` als zweites Element, während hingegen das zweite Element der linken Liste `xs::[Int]` ist. Weiterhin ist das erste Element der linken Liste `x:ys::[Int]` und das erste der rechten Liste `x:ys ++ xs::[Int]`. Die Listen sind also verschieden.

e) Beide Ausdrücke sind korrekt getypt und entsprechen Listen vom Typ `[[Int]]`. Die linke Liste hat Länge 2, während die rechte Liste die Länge 3 besitzt.

Beide Ausdrücke entsprechen daher verschiedenen Listen.

Tutoraufgabe 7 (Programmieren in Haskell):

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Verwenden Sie außer den Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise), `True` und `False`, Werten des Typs `Int`, der Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., booleschen Funktionen wie `&&`, `||`, `not` und arithmetischen Operatoren wie `+`, `*` **keine** vordefinierten Funktionen außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden. Schreiben Sie ggf. Hilfsfunktionen, um sich die Lösung der Aufgaben zu vereinfachen.

a) `fib n`
Berechnet die `n`-te Fibonacci–Zahl. Auf negativen Eingaben darf sich die Funktion beliebig verhalten.

Die Auswertung von `fib 17` liefert bspw. den Ausgabewert `1597`.

Hinweise:

- Die Fibonacci–Zahlen sind durch die rekursive Folge mit den Werten $a_0 = 0$, $a_1 = 1$ und $a_n = a_{n-2} + a_{n-1}$ für $n \geq 2$ beschrieben.

b) `prime n`
Gibt genau dann `True` zurück, wenn die natürliche Zahl `n` eine Primzahl ist. Auf negativen Eingaben darf sich die Funktion beliebig verhalten.

Die Auswertung von `prime 35897` liefert bspw. den Ausgabewert `True`.

Hinweise:

- Sie dürfen die vordefinierte Funktion `rem x y` verwenden, die den Rest der Division `x` / `y` zurückgibt.

c) `nthSmallestPerfectNumber n`
Diese Funktion soll die `n`. kleinste vollkommene (engl. perfect) Zahl berechnen. Bei nicht-positiver Eingabe `n` darf sich die Funktion beliebig verhalten. Eine natürliche Zahl heißt vollkommen, wenn diese die Summe aller ihrer echten Teiler ist, d.h. aller Teiler außer sich selbst. Die kleinsten vollkommenen Zahlen sind $6 = 1 + 2 + 3$, $28 = 1 + 2 + 4 + 7 + 14$ und $496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248$.

Entsprechend sollen die Ausdrücke `nthSmallestPerfectNumber 1`, `nthSmallestPerfectNumber 2` und `nthSmallestPerfectNumber 3` zu 6, 28 und 496 auswerten.

d) `powersOfTwo i0 i1`
Gibt eine `Int`–Liste zurück, die die Zweierpotenzen $2^{\texttt{i0}}$ bis $2^{\texttt{i1}}$ enthält. Falls `i0` $>$ `i1`, soll die leere Liste zurückgegeben werden. Auf negativen Eingaben darf sich die Funktion beliebig verhalten.

Die Auswertung von `powersOfTwo 5 10` liefert bspw. den Ausgabewert `[32,64,128,256,512,1024]`.

Hinweise:

- Sie können die Exponentiation x^y zweier Zahlen `x` und `y` in Haskell mit `x^y` vornehmen.

e) `selectKsmallest k xs`
Gibt das Element zurück, das in der `Int`-Liste `xs` an der Stelle `k` stehen würde, wenn man `xs` aufsteigend sortiert. Hierbei hat das erste Element den Index 1. Wenn `k` kleiner als 1 oder größer als die Länge von `xs` ist, darf sich die Funktion beliebig verhalten.

Die Auswertung von `selectKsmallest 3 [4, 2, 15, -3, 5]` liefert also den Ausgabewert `4` und `selectKsmallest 1 [5, 17, 1, 3, 9]` liefert den Ausgabewert `1`.

Hinweise:

- Sie können die Liste an einem geeigneten Element `x` in zwei Listen teilen, sodass eine der beiden Teillisten nur Elemente enthält, die kleiner oder gleich `x` sind, und die andere Teilliste nur größere Elemente als `x` enthält. Dann können Sie `selectKsmallest` mit geeigneten Parametern rekursiv aufrufen.
- Sie dürfen die vordefinierte Funktion `length` verwenden, wobei `length ys` die Anzahl der Elemente der Liste `ys` zurückgibt.

Lösung:

a)

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

b)
```
prime :: Int -> Bool
prime 0 = False
prime 1 = False
prime n = isPrime (properDivisors n 1)
  where
    isPrime            (_:_:_) = False
    isPrime            _       = True

properDivisors :: Int -> Int -> [Int]
properDivisors n m | n <= m         = []
                   | rem n m == 0   = m:properDivisors n (m+1)
                   | otherwise      = properDivisors n (m+1)
```

c)
```
nthSmallestPerfectNumber :: Int -> Int
nthSmallestPerfectNumber n = tryAll n 1
  where
    sumList []     = 0
    sumList (x:xs) = x + sumList xs
    tryAll n k | sumList (properDivisors k 1) == k && n > 1
                 = tryAll (n-1) (k+1)
               | sumList (properDivisors k 1) == k
                 = k
               | otherwise
                 = tryAll n (k+1)
```

d)
```
powersOfTwo :: Int -> Int -> [Int]
powersOfTwo n m | n > m   = []
                | otherwise = (2^n) : powersOfTwo (n+1) m
```

e)
```
selectKsmallest :: Int -> [Int] -> Int
selectKsmallest _ [] = 0
selectKsmallest k (pivot:rest) =
  let
  split :: [Int] -> ([Int], [Int])
  split [] = ([], [])
  split (y:ys) = if y <= pivot then (y:left, right) else (left, y:right)
                 where (left, right) = split ys
  left, right :: [Int]
  (left, right) = split rest
  leftLen :: Int
  leftLen = length left
  in
   if leftLen == (k-1) then pivot else
     if leftLen > (k-1) then selectKsmallest k left else
       selectKsmallest (k-1-leftLen) right
```

Tutoraufgabe 8 (Programmieren in Haskell (Video)):

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Sie dürfen die Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise), `True` und `False`, Werte des Typs `Int`, die Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., boolesche Funktionen wie `&&`, `||`, `not` und die arithmetischen Operatoren `+`, `*`, `-` verwenden, aber **keine** vordefinierten Funktionen außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden. Schreiben Sie ggf. Hilfsfunktionen, um sich die Lösung der Aufgaben zu vereinfachen.

a) `fibInit a0 a1 n`
Berechnet die `n`-te Fibonacci–Zahl der Fibonacci-Folge mit den alternativen natürlichen Initialwerten `a0` und `a1`. Auf negativen Eingaben für `a0`, `a1` und `n` darf sich die Funktion beliebig verhalten.

Die Auswertung von `fibInit 1 11 7` liefert bspw. den Rückgabewert `151`.

Hinweise:

- Die Fibonacci–Zahlen mit den Initialwerten `a0` und `a1` sind durch die rekursive Folge mit den Werten $a_0 =$ `a0`, $a_1 =$ `a1` und $a_n = a_{n-2} + a_{n-1}$ für $n \geq 2$ beschrieben.

b) In Aufgabe 7 wurden die Fibonacci–Zahlen mit einer naiven Implementierung berechnet, die schon für $n = 50$ nicht in annehmbarer Zeit zu einem Ergebnis kommt. Das liegt daran, dass für den Aufruf `fib 50` sowohl `fib 49` als auch `fib 48` ausgwertet werden müssen. Für `fib 49` muss aber auch `fib 48` (und `fib 47`) ausgewertet werden. Offensichtlich berechnen wir so dasselbe Ergebnis — `fib 48` — mehrmals. In der Implementierung in dieser Aufgabe soll dieser Mehraufwand umgangen werden.

Dazu gehen wir schrittweise vor: Implementieren Sie zunächst die Funktion `fibInitL a0 a1 n`. Diese berechnet die Liste der nullten, ersten, ..., $(n-1)$-ten, n-ten Fibonacci–Zahl der Fibonacci-Folge mit den alternativen natürlichen Initialwerten `a0` und `a1` auf effiziente Art und Weise. Falls $n = -1$ ist, so liefert `fibInitL a0 a1 (-1)` das Resultat `[]`. Für $n < -1$ darf sich die Funktion beliebig verhalten.

Die Auswertung von `fibInitL 0 1 6` liefert bspw. den Rückgabewert `[0,1,1,2,3,5,8]`.

Implementieren Sie dann die Funktion `fibInit2 a0 a1 n`, die die `n`-te Fibonacci–Zahl der Fibonacci-Folge mit den alternativen natürlichen Initialwerten `a0` und `a1` auf effiziente Art und Weise berechnet. Auf negativen Eingaben für `a0`, `a1` und `n` darf sich die Funktion beliebig verhalten.

Die Auswertung von `fibInit2 1 3 50` liefert bspw. den Rückgabewert `45537549124`.

Hinweise:

- Sie können mit obigem Beispiel überprüfen, ob ihre Implementierung effizient ist.

c) `normalize xs`
Gibt eine `Int`–Liste von derselben Länge wie `xs` zurück, deren kleinster Wert 0 ist. Weiterhin soll die Differenz zwischen zwei benachbarten Zahlen in der Ausgabeliste stets genauso hoch sein wie die Differenz zwischen den beiden Zahlen der Eingabeliste an denselben Positionen.

Die Auswertung von `normalize [15,17,-3,46]` liefert bspw. den Rückgabewert `[18,20,0,49]`.

d) `sumMaxs xs`
Addiert diejenigen Werte der eingegebenen `Int`–Liste `xs` auf, die größer sind als **alle** vorherigen Werte in der Liste.

Die Auswertung von `sumMaxs [2,1,2,5,4]` liefert bspw. den Rückgabewert `7` $(= 2 + 5)$.

e) `sumNonMins xs`
Addiert diejenigen Werte der eingegebenen `Int`–Liste `xs` auf, die größer sind als mindestens **ein** vorheriger Wert in der Liste.

Die Auswertung von `sumNonMins [2,1,2,5,4]` liefert bspw. den Rückgabewert `11` $(= 2 + 5 + 4)$.

f) `primeTwins x`
Gibt den kleinsten Primzahl–Zwilling zurück, dessen beide Elemente größer sind als die `Int`-Zahl `x`.

Die Auswertung von `primeTwins 12` liefert bspw. den Rückgabewert `(17,19)`.

Hinweise:

- Ein Primzahlzwilling ist ein 2-Tupel $(n, n + 2)$, bei dem sowohl n als auch $n + 2$ Primzahlen sind.
- Sie dürfen die Funktion `prime` aus Aufgabe 7 verwenden.

g) `multiples xs i0 i1`
Gibt eine `Int`–Liste zurück, die alle Werte zwischen `i0` und `i1` enthält, die ein Vielfaches einer der Werte aus `xs` sind. Die zurückgegebene Liste soll die Werte in aufsteigender Reihenfolge und jeweils nur einmal enthalten.

Die Auswertung von `multiples [3,5] 5 20` liefert bspw. den Rückgabewert `[5,6,9,10,12,15,18,20]`.

Hinweise:

- Sie dürfen die vordefinierte Funktion `rem x y` verwenden, die den Rest der Division `x` / `y` zurückgibt.

Lösung:

a)
```
fibInit :: Int -> Int -> Int -> Int
fibInit n0 _ 0 = n0
fibInit _ n1 1 = n1
fibInit n0 n1 n = fibInit n0 n1 (n-2) + fibInit n0 n1 (n-1)
```

b)
```
fibInitL :: Int -> Int -> Int -> [Int]
fibInitL _ _ (-1) = []
fibInitL n0 n1 n = n0 : fibInitL n1 (n0+n1) (n-1)

fibInit2 :: Int -> Int -> Int -> Int
fibInit2 n0 n1 n = lastHelp (fibInitL n0 n1 n)

lastHelp :: [Int] -> Int
lastHelp [x] = x
lastHelp (x : xs) = lastHelp xs
```

c)
```
normalize :: [Int] -> [Int]
normalize [] = []
normalize xs = subt xs (minHelp xs)

minHelp :: [Int] -> Int
minHelp [x] = x
minHelp (x : y : xs) | x < y = minHelp (x : xs)
                     | otherwise = minHelp (y : xs)

subt :: [Int] -> Int -> [Int]
subt [] _ = []
subt (x : xs) y = (x - y) : subt xs y
```

d)
```
sumMaxs :: [Int] -> Int
sumMaxs [] = 0
sumMaxs (x : xs) = x + sumMaxsHelp x xs

sumMaxsHelp :: Int -> [Int] -> Int
sumMaxsHelp x [] = 0
sumMaxsHelp x (y : ys) | y > x = y + sumMaxsHelp y ys
                       | otherwise = sumMaxsHelp x ys
```

e)
```
sumNonMins :: [Int] -> Int
sumNonMins [] = 0
sumNonMins (x : xs) = sumNonMinsHelp x xs

sumNonMinsHelp :: Int -> [Int] -> Int
sumNonMinsHelp x [] = 0
sumNonMinsHelp x (y : ys) | y > x = y + sumNonMinsHelp x ys
                          | otherwise = sumNonMinsHelp y ys
```

f)
```
primeTwins :: Int -> (Int, Int)
primeTwins n | prime (n+1) && prime (n+3) = (n+1, n+3)
             | otherwise = primeTwins (n+1)

-- prime von Aufgabe 7
prime :: Int -> Bool
prime 0 = False
prime 1 = False
prime n = isPrime (properDivisors n 1)
  where
    isPrime            (_:_:_) = False
    isPrime            _       = True

properDivisors :: Int -> Int -> [Int]
properDivisors n m | n <= m        = []
                   | rem n m == 0  = m:properDivisors n (m+1)
                   | otherwise     = properDivisors n (m+1)
```

g)
```
multiples :: [Int] -> Int -> Int -> [Int]
multiples [] _ _ = []
multiples (x : xs) n m | n /= m = multiples (x : xs) n n
                                  ++ multiples (x : xs) (n+1) m
                       | n == m && rem n x == 0 = n : []
                       | n == m && not (rem n x == 0) = multiples xs n m
```

Aufgabe 9 (Programmieren in Haskell): (7+6+10+9+11+7 = 50 Punkte)

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Sie dürfen die Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise), `True` und `False`, Werte des Typs `Int`, die Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., boolesche Funktionen wie `&&`, `||`, `not` und die arithmetischen Operatoren `+`, `*`, `-` verwenden, aber **keine** weiteren vordefinierten Funktionen. Schreiben Sie ggf. Hilfsfunktionen, um sich die Lösung der Aufgaben zu vereinfachen. Sie dürfen jederzeit Hilfsfunktionen aus vorherigen Teilaufgaben verwenden, auch wenn Sie diese nicht selbst implementiert haben.

a) `symmetricDifference xs ys`
Diese Funktion berechnet die symmetrische Differenz zweier Listen `xs::[Int]` und `ys::[Int]`. Die symmetrische Differenz ist eine Liste `zs::[Int]`, welche alle Elemente enthält, die entweder nur in `xs` oder nur in `ys` enthalten sind. Die Reihenfolge der Elemente innerhalb der resultierenden Liste ist hierbei unerheblich. Wenn ein Wert sowohl in `xs` als auch in `ys` vorkommt, darf er nicht in `symmetricDifference xs ys` auftreten, auch wenn er unterschiedlich oft in `xs` und `ys` vorkommt.

Die Auswertung von `symmetricDifference [1,1,2,4,5] [2,2,3,4,6]` kann beispielsweise die Liste `[1,1,5,3,6]::[Int]` liefern.

b) `powerlist xs`
Berechnet eine Liste `ps::[[Int]]` aller Teillisten von `xs::[Int]`. Für jede Liste in `ps` soll die Reihenfolge der Listenelemente dieselbe sein wie die Reihenfolge der entsprechenden Elemente in `xs`. Die Reihenfolge der Elemente von `ps` selbst ist hierbei unerheblich. Mehrfach vorkommende Werte in `xs` können auch in den Teillisten entsprechend oft auftreten. Wenn l die Länge von `xs` bezeichnet, so ist 2^l die Länge von `powerlist xs`.

Die Auswertung von `powerlist [1,2,3]` kann beispielsweise die Liste `[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]` der Länge 8 ergeben. Der Ausdruck `powerlist [2,2]` kann hingegen beispielsweise zur Liste `[[],[2],[2],[2,2]]` ausgewertet werden.

c) `permutations xs`
Diese Funktion soll zu einer Liste `xs::[Int]` eine Liste aller Permutationen von `xs` berechnen. Eine Permutation von `xs` ist hierbei eine Liste `ps::[Int]`, welche dieselben Elemente wie `xs` enthält, jedoch kann die Reihenfolge der Elemente in `ps` von der Reihenfolge der Elemente in `xs` abweichen. Die Reihenfolge der Elemente von `permutations xs` kann hierbei beliebig sein. Wenn `xs` eine Liste der Länge l ist, so ist `permutations xs` von der Länge $l!$. Falls `xs` die leere Liste ist, so darf sich die Funktion beliebig verhalten.

Der Ausdruck `permutations [1,2,3]` könnte damit zu der Liste `[[1,2,3],[2,1,3],[3,2,1],[1,3,2],[3,1,2],[2,3,1]]` vom Typ `[Int]` auswerten.

Die folgenden Teilaufgaben beziehen sich auf eine Datenstruktur, welche Graphen repräsentiert. Mathematisch ist ein Graph ein Tupel $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, wobei $\mathcal{V}$ eine Menge von Knoten (engl. vertices) und $\mathcal{E} \subseteq \mathcal{V}^2$ eine Menge von Kanten (engl. edges) ist, welche die Knoten untereinander verbinden. Die betrachteten Graphen sind gerichtet, d.h., eine Kante (a, b) bedeutet, dass Knoten b von Knoten a erreicht werden kann, jedoch nicht unbedingt auch umgekehrt. Wir nehmen im Folgenden an, dass jeder Knoten mindestens eine eingehende oder ausgehende Kante besitzt.

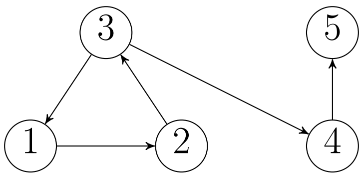

Abbildung 1: Durch die Liste `testGraph = [(1,2),(2,3),(3,1),(4,5),(3,4)]::[(Int,Int)]` repräsentierter Graph.

Abb. 1 ist eine graphische Repräsentation des Graphen $(\{1, 2, 3, 4, 5\}, \{(1, 2), (2, 3), (3, 1), (4, 5), (3, 4)\})$. Im Folgenden werden Graphen in Haskell als Liste vom Typ `[(Int,Int)]` ihrer Kanten dargestellt. Der in Abb. 1 abgebildete Graph kann als Liste `testGraph = [(1,2),(2,3),(3,1),(4,5),(3,4)]::[(Int,Int)]` seiner Kanten in Haskell dargestellt werden.

d) `nodes es`
Gegeben eine Liste `es::[(Int,Int)]` von Kanten, berechnet diese Funktion eine Liste vom Typ `[Int]` aller im Graph enthaltenen Knoten. Die berechnete Liste soll *keine Duplikate* enthalten. Die Reihenfolge ist irrelevant.

Beispielsweise könnte `nodes testGraph` zu der Liste `[1,2,3,4,5]::[Int]` auswerten.

e) `existsPath es a b`
Diese Funktion berechnet für eine gegebene Liste `es::[(Int,Int)]` von Kanten und zwei Knoten `a,b::Int` einen Wahrheitswert vom Typ `Bool`, der angibt, ob der Knoten `b` vom Knoten `a` aus mithilfe der Kanten aus der Liste `es` erreicht werden kann.

Die Ausdrücke `existsPath testGraph 1 3`, `existsPath testGraph 1 5` und `existsPath testGraph 5 5` berechnen hierbei beispielsweise den Wahrheitswert `True`, während hingegen die Aufrufe `existsPath testGraph 5 1`, `existsPath testGraph 5 4` und `existsPath testGraph 4 3` zu `False` auswerten. Für jeden Knoten `a` gilt hierbei, dass `existsPath es a a` zu `True` auswertet, da man den Knoten `a` immer von sich selbst aus über einen Pfad aus 0 Kanten erreichen kann.

Hinweise:

- Überlegen Sie, wie die Liste der Kanten des Graphen in einem rekursiven Aufruf geeignet modifiziert werden kann, um Terminierung bei zyklischen Graphen sicherzustellen.

f) `isConnected es`
Diese Funktion berechnet, ob ein durch eine Liste von Kanten `es::[(Int,Int)]` dargestellter Graph zusammenhängend ist. Ein Graph heißt zusammenhängend, wenn für alle Knoten `a` und `b` der Knoten `b` von `a` aus erreichbar ist, d.h., `existsPath es a b` ist `True` für alle Knoten `a` und `b`.

Zum Beispiel wertet `isConnected testGraph` zu `False` und `isConnected ((5,1):testGraph)` zu `True` aus.

Lösung:

a)
```
remove :: Int -> [Int] -> [Int]
remove _ [] = []
remove e (x:xs) = if e==x then remove e xs else x:remove e xs

symmetricDifference :: [Int] -> [Int] -> [Int]
symmetricDifference xs ys = difference xs ys ++ difference ys xs
  where
    difference :: [Int] -> [Int] -> [Int]
    difference xs []     = xs
    difference xs (y:ys) = difference (remove y xs) ys
```

b)
```
powerlist :: [Int] -> [[Int]]
powerlist [] = [[]]
powerlist (x:xs) = powerlist xs ++ insertFront (powerlist xs)
  where
    insertFront :: [[Int]] -> [[Int]]
    insertFront []     = []
    insertFront (p:ps) = (x:p):insertFront ps
```

c)
```
permutations :: [Int] -> [[Int]]
permutations []     = [[]]
permutations [x]    = [[x]]
permutations (x:xs) = helper (permutations xs)
  where
    insertEverywhere :: [Int] -> [Int] -> [[Int]]
    insertEverywhere ps []     = [ps++[x]]
    insertEverywhere ps (e:es) =
      (ps++(x:e:es)):insertEverywhere (ps++[e]) es
    helper :: [[Int]] -> [[Int]]
    helper []     = []
    helper (l:ls) = insertEverywhere [] l ++ helper ls
```

d)
```
testGraph = [(1,2),(2,3),(3,1),(4,5),(3,4)] :: [(Int,Int)]
nodes :: [(Int,Int)] -> [Int]
nodes xs = removeDuplicates (allNodes xs)
  where
    allNodes :: [(Int,Int)] -> [Int]
    allNodes [] = []
    allNodes ((a,b):es) = a:b:allNodes es
    removeDuplicates :: [Int] -> [Int]
    removeDuplicates [] = []
    removeDuplicates (x:xs) = x:removeDuplicates (remove x xs) -- a)
```

e)
```
existsPath :: [(Int,Int)] -> Int -> Int -> Bool
existsPath g a b = helper [] a b g
  where
    helper :: [(Int,Int)] -> Int -> Int -> [(Int,Int)] -> Bool
    helper _  a b [] = a == b
    helper us a b ((x,y):es)
      | a == x    = existsPath (us++es) y b || helper us a b es
      | otherwise = helper ((x,y):us) a b es
```

f)
```
isConnected :: [(Int,Int)] -> Bool
isConnected g = checkPairs vs vs
  where
    vs :: [Int]
    vs = nodes g
    checkPairs :: [Int] -> [Int] -> Bool
    checkPairs [] _ = True
    checkPairs (_:xs) [] = checkPairs xs vs
    checkPairs (x:xs) (y:ys) = existsPath g x y && checkPairs (x:xs) ys
```