```
data Bit = One | Zero deriving Show
type Bits = [Bit]

integer2Bits :: Integer -> Bits
integer2Bits 1 = [One]
integer2Bits 0 = [Zero]
integer2Bits x = integer2Bits (div x 2) ++ integer2Bits(mod x 2)

bits2String :: Bits -> String
bits2String [One] = "One"
bits2String [Zero] = "Zero"
bits2String (x:xs) = bits2String[x] ++ bits2String(xs)

bits2Integer :: Bits -> Integer
bits2Integer [One] = 1
bits2Integer [Zero] = 0
bits2Integer (x:xs) = bitString2Integer (bits2String (x:xs))

bitString2Integer :: String -> Integer
bitString2Integer "1" = 1
bitString2Integer "0" = 0
bitString2Integer (x:xs) = (2^(length(x:xs) -1) ) * bitString2Integer [x] + bitString2Integer xs

integer2binString :: Integer -> String
integer2binString 1 = "One"
integer2binString 0 = "Zero"
integer2binString x = integer2binString (div x 2 ) ++ integer2binString (mod x 2 )

--1
istLeer :: [a] -> Bool
istLeer [] = True
istLeer (x:xs) = False

--2
snoc :: [a] -> a -> [a]
snoc [] a = [a]
snoc (x:xs) a = x:(snoc xs a)

--3
laenge :: [a] -> Integer
laenge [] = 0
laenge (_:xs) = 1 + laenge xs

--4
erstesElement :: [a] -> a
erstesElement [] = error "Liste leer"
erstesElement (x:_) = x

--5
rest :: [a] -> [a]
rest (x:xs) = xs
```

```
--6
letztesElement :: [a] -> a
letztesElement [] = error "Liste leer"
letztesElement [x] = x
letztesElement (_:xs) = letztesElement xs

--7
anfang :: [a] -> [a]
anfang [] = error "Liste leer"
anfang (x:xs) = [x]

--8
nimm :: Int -> [a] -> [a]
nimm 0 _ = []
nimm n [] = []
nimm n(x:xs) = x:(nimm(n-1)xs)

--9
verwerfe :: Int -> [a] -> [a]
verwerfe 0 xs = xs
verwerfe _ [] = []
verwerfe n (_:xs) = verwerfe (n-1) xs

--10
summe :: (Num a) => [a] -> a
summe xs = sum xs

--11
verdopple :: (Num a) => [a] -> [a]
verdopple (x:xs) = (2 * x) : verdopple xs

--12
verkette :: [a] -> [a] -> [a]
verkette xs ys = (xs ++ ys)

--13
rueckwaerts :: [a] -> [a]
rueckwaerts [] = []
rueckwaerts (x:xs) = rueckwaerts xs ++ [x]

--14
und :: [Bool] -> Bool
und [] = True
und (x:xs)
        | x == False = False
        | otherwise = und xs

--15
oder :: [Bool] -> Bool
oder [] = True
oder (x:xs)
        | x == True = True
        | otherwise = False

--16
aufteilen :: Int -> [a] -> ([a],[a])
aufteilen n xs = splitAt n xs

--17
verzahne :: [a] -> [b] -> [(a,b)]
verzahne [] _ = []
verzahne (x:xs) (y:ys) = (x, y) : verzahne xs ys

--18
--aktualisiere :: [a] -> Integer -> a -> [a]
--aktualisiere xs n e = splitAt xs n
```