Tuan

Question 11.1:

```
> foldBool :: a -> a -> Bool -> a
> foldBool x y bool
>  | bool = x
>  | otherwise = y

> data Day = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday
| Saturday
>  deriving (Eq, Show)

> foldDay :: a -> a -> a -> a -> a -> a -> a -> Day -> a
> foldDay sun mon tue wed thu fri sat day
>  | day == Sunday    = sun
>  | day == Monday    = mon
>  | day == Tuesday   = tue
>  | day == Wednesday = wed
>  | day == Thursday  = thu
>  | day == Friday    = fri
>  | day == Saturday  = sat
```

Question 11.2:

```
> f :: Bool -> Bool -> Bool
> f a b = not a || b
```

Question 11.3:

```
> data Set a = Empty | Singleton a | Union (Set a) (Set a)
>  deriving Show

> foldSet :: (b -> b -> b) -> (a -> b) -> b -> Set a -> b
> foldSet union singleton empty Empty = empty
> foldSet union singleton empty (Singleton a) = singleton a
> foldSet union singleton empty (Union xs ys) = union (sublistFold xs)
(sublistFold ys)
>  where sublistFold = foldSet union singleton empty

> isIn :: Eq a => a -> Set a -> Bool
> isIn n = foldSet (||) (==n) False

> setToList :: Eq a => Set a -> [a]
> setToList = foldSet (++) toList []
>  where toList n = [n]
```

```
> subset :: Eq a => Set a -> Set a -> Bool
> subset xs ys = and (map (isIn2 ys) (setToList xs))
>  where isIn2 ys x = isIn x ys

> instance Eq a => Eq (Set a) where
>  xs == ys = (xs `subset` ys) && (ys `subset` xs)
```

Question 11.4:

```
> data Btree a = Leaf a | Fork (Btree a) (Btree a)
>  deriving Show

> data Direction = L | R
>  deriving (Eq, Show)

> type Path = [Direction]

> foldBtree :: (b -> b -> b) -> (a -> b) -> Btree a -> b
> foldBtree fork leaf (Leaf a) = leaf a
> foldBtree fork leaf (Fork l r) = fork (subfold l) (subfold r)
>  where subfold = foldBtree fork leaf

> isIn' :: Eq a => a -> Btree a -> Bool
> isIn' n = foldBtree (||) (==n)

> find :: Eq a => a -> Btree a -> Maybe Path
> find n t = if path /= [] then Just path else Nothing
>  where path = findAuxilary n t

> findAuxilary :: Eq a => a -> Btree a -> Path
> findAuxilary n (Leaf t) = []
> findAuxilary n (Fork l r)
>  | isIn' n l = [L] ++ findAuxilary n l
>  | isIn' n r = [R] ++ findAuxilary n r
>  | otherwise = []
```

Question 12.1:

```
> type Queue a = [a]

> empty :: Queue a
> empty = []
```

```
> isEmpty :: Queue a -> Bool
> isEmpty queue = null queue

> add :: a -> Queue a -> Queue a
> add n queue = n : queue

> get :: Queue a -> (a, Queue a)
> get queue = (last queue, init queue)
```

The *empty* function takes a constant amount of time. The *isEmpty* function also takes a constant amount of time. *Add* takes O(n) because of (++), but *get* takes O(1) steps to get to the last element of the queue. If the queue was represented by a list of its elements in the reverse order, the functions *empty*, *isEmpty* would take the same time, but *add* would be O(1), and *get* would be O(n). The alternative implementation of queues is:

```
> type Queue' a = ([a], [a])

> valid :: Queue' a -> Bool
> valid (xs, ys) = not (null xs) || null ys

> empty' :: Queue' a
> empty' = ([], [])

> isEmpty' :: Queue' a -> Bool
> isEmpty' (xs, ys) = null xs

> add' :: a -> Queue' a -> Queue' a
> add' x (xs, ys) = if null xs then (reverse ys, []) else (xs, x : ys)

> get' :: Queue' a -> (a, Queue' a)
> get' queue = (head (fst queue),
>               ([head (snd queue)], tail $ snd queue))
```

With this implementation all the functions are O(1).

<u>Question 12.2:</u>

With the normal recursive function as n becomes bigger the calls become slower because the smaller values have to be calculated multiple times. For example *fib 4 = fib 3 + fib 2 = 3 fib 1 + 2 fib 0.*

```
> sumTuple :: (Integer, Integer) -> Integer
> sumTuple (a,b) = a + b
```

```
> two :: Integer -> (Integer, Integer)
> two 0 = (0, 1)
> two n = (snd (two (n-1)), sumTuple $ two $ n-1)

> fib' :: Integer -> Integer
> fib' n = fst (two n)
```

With this function it takes roughly n steps to calculate *fib n.*

I will prove with induction on n that $F^n = \begin{matrix} fib\ (n-1) & fib\ n \\ fib\ n & fib\ (n+1) \end{matrix}$.
Base case: $F^0 = \begin{matrix} 0 & 1 \\ 1 & 1 \end{matrix}$, so the claim is true for the base case. For the inductive step let us assume that $F^n = \begin{matrix} fib\ (n-1) & fib\ n \\ fib\ n & fib\ (n+1) \end{matrix}$. We will now have to prove for *(n+1).*

$$F^{n+1} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} fib\ (n-1) & fib\ n \\ fib\ n & fib\ (n+1) \end{pmatrix} (IH) = \begin{pmatrix} fib\ n & fib\ (n+1) \\ fib\ (n+1) & fib\ (n+2) \end{pmatrix}$$

Therefore, the claim is true for all natural numbers.

```
> mmult :: Num a => [[a]] -> [[a]] -> [[a]]
> mmult a b =
>       [[ sum $ zipWith (*) ar bc | bc <- (transpose b)] | ar <- a]

> powerM :: [[Integer]] -> [[Integer]] -> Integer -> [[Integer]]
> powerM y x n
>         | n == 0 = y
>         | even n = powerM y (x `mmult` x) (n `div` 2)
>         | odd n  = powerM (x `mmult` y) x (n-1)

> fib'' :: Integer -> Integer
> fib'' n = (head (powerM y x n)) !! 1
>  where y = [[1, 0], [0, 1]]
>        x = [[0, 1], [1, 1]]
```