Oplossingen Week 2

```
module Week2 where
import Data.List (group)
```

Grootste gemene deler

Een recursieve oplossing is hier het makkelijkst.

```
ggd :: Int -> Int -> Int
ggd a 0 = a
ggd a b = ggd b $ a `mod` b
```

CoPrime

We kunnen in deze oplossing gebruik maken van bovenstaande `ggd`.

```
coprime :: Int -> Int -> Bool
coprime x y = ggd x y == 1
```

Priem of niet?

Om te kijken of een getal een priemgetal is, kijken we dus gewoon of het ondeelbaar is door alle kleinere getallen groter dan 1. Een eerste mogelijkheid is om een lijst op te stellen van alle niet-triviale delers met een LC. Vervolgens kijken we naar de lengte van die lijst: is ze leeg, hebben we te maken met een priemgetal.

```
isPriemgetal :: Int -> Bool
isPriemgetal n = null [ x | x <- [2..n - 1], n `mod` x == 0 ]
```

Deze oplossing zal ook werken voor 0 en 1, omdat `[2..0]` en `[2..1]` gewoon leeg zijn.

Anders is het ook mogelijk om gebruik te maken van de ingebouwde `all` methode. Deze zal voor alle elementen in een lijst kijken of de gegeven functie waar blijkt.

```
isPriemgetal' :: Int -> Bool
isPriemgetal' n = all undiv [2..n - 1]
  where undiv x = n `mod` x /= 0
```

Wie efficiëntie nastreeft, zal bij voorkeur maar controleren tot de vierkantswortel van gegeven getal. We kunnen hier gebruik maken van de `sqrt :: Floating a => a -> a` functie. Deze verwacht echter een `Floating a` als argument. We kunnen onze `Int` omzetten naar een algemeen `Num a` met de `fromIntegral :: (Num b, Integral a) => a -> b` functie.

```haskell
root :: Int -> Int
root = floor . sqrt . fromIntegral
```

Om alle priemgetallen te berekenen, worden we gevraagd om gebruik te maken van `isPriemgetal` en LCs.

```haskell
allePriemgetallen :: [Int]
allePriemgetallen = filter isPriemgetal [2..]
```

Tenslotte kunnen we `geefPriemgetallen` schrijven, die gewoon een aantal elementen vooraan `allePriemgetallen` neemt.

```haskell
geefPriemgetallen :: Int -> [Int]
geefPriemgetallen n = take n allePriemgetallen
```

Wie eens `allePriemgetallen !! 10000` uitvoerde, zal al gemerkt hebben dat hij even moet wachten. Als bonus komt hieronder nog een efficiëntere manier om alle priemgetallen tot een zekere `n` te berekenen, namelijk met behulp van een variant op de Zeef van Eratosthenes.

```haskell
-- x is niet deelbaar door y
(%%) :: Int -> Int -> Bool
x %% y = x `mod` y /= 0

geefPriemgetallen' :: Int -> [Int]
geefPriemgetallen' n = loop [2..n]
  where loop []                 = []
        loop (p:xs) | p * p > n = p : xs
                    | otherwise = p : loop [ x | x <- xs, x %% p ]
```

We filteren hier recursief een lijst van kandidaten. Bij elke aanroep zal het eerste getal uit de lijst het volgende priemgetal zijn. Dan filteren we alle veelvouden van dit priemgetal uit de rest van de lijst. Als het kwadraat van ons priemgetal groter is dan `n`, hoeven we niet te filteren; tenslotte zijn alle kleinere veelvouden al gefilterd.

Ik demonstreer hier ook hoe we eigen operatoren kunnen definiëren op dezelfde manier als andere functies. Het enige verschil is dat functies enkel bestaande uit "tekentjes" standaard infix gebruikt worden.

Als extraatje demonstreer ik nog een efficiënter manier om deze drie functies tesamen te schrijven, waarbij ze elkaar recursief aanroepen. De `primes` functie filtert de oneindige lijst `[3..]` met de `isPrime` functie. De `isPrime` functie geeft terug of een gegeven getal een priemgetal is door het te proberen delen door alle priemgetallen kleiner dan zijn vierkantswortel, gegeven door `smallerPrimes`. De functie `smallerPrimes` neemt alle priemgetallen kleiner dan de wortel van gegeven getal uit de lijst van alle priemgetallen, `primes`.

```haskell
primes :: [Int]
primes = 2 : filter isPrime [3..]
```

```
isPrime :: Int -> Bool
isPrime n = all (n %%) $ smallerPrimes n

smallerPrimes :: Int -> [Int]
smallerPrimes n = takeWhile (\i -> i*i <= n) primes
```

Dit hele truukje werkt omdat we de recursie stoppen door 1 priemgetal vast te definiëren, namelijk 2. Hieronder de stappen die gebeuren als we het 2de priemgetal (`primes !! 1 == 3`) opvragen, telkens met de huidige staat van `primes` in het geheugen vermeld. We gebruiken hier de (vereenvoudigde) recursieve definitie van `!!`:

```
[]     !! _ = fail "index out of bounds"
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n - 1)
```

`primes !! 1` wordt gematcht met de definities van `!!`. We hebben meer informatie over de lijst nodig, dus rekenen ze 1 stap verder uit. `(2 : filter isPrime [3..]) !! 1` matcht met de derde definitie, we passen deze toe. In het geheugen zit nu dat `primes = 2 : filter isPrime [3..]`. We rekenen verder met `(filter isPrime [3..]) !! 0`. Dit zou matchen met de tweede regel van `!!`, maar we moeten de eerste term verder uitrekenen. `isPrime 3` wordt `all (3 %%) $ smallerPrimes n`. `smallerPrimes n` wordt `takeWhile (\i -> i*i <= n) (2:...)`. Omdat `2*2 > 3` zal takeWhile een lege lijst teruggeven **zonder** naar de rest van `primes` te kijken. `all (3 %%) []` is triviaal waar, dus `isPrime 3` is ook `True`. Onze `filter isPrime [3..]` wordt bijgevolg `3 : filter isPrime [4..]`. We kunnen de tweede regel van `!!` met succes toepassen, en het antwoord is 3.

Algemeen, omdat onze `smallerPrimes` enkel priemgetallen opvraagt *kleiner* dan hetgene we momenteel controleren, en die kleinere priemgetallen eerder al uitgerekend zijn, zal deze functie altijd een waarde kunnen teruggeven. Op die manier wordt de recursie-cirkel doorbroken.

Naderen tot Pi

We zouden Pi recursief kunnen berekenen, maar kiezen hier om de ingebouwde `sum` te gebruiken.

```
berekenPi :: Integer -> Double
berekenPi i = (*4) . sum $ map (term . fromInteger) [0..i]
  where term n = (-1)**n / (2*n + 1)
```

Elementen groeperen

We kunnen dit recursief oplossen.

```haskell
pack :: Eq a => [a] -> [[a]]
pack []     = []
pack (l:ls) = map reverse $ rec [l] ls
  where rec (x:xs) (y:ys) | x == y    = rec (y:x:xs) ys
                          | otherwise = (x:xs) : rec [y] ys
        rec xs     []                 = [xs]
        rec []     _                  = fail "unreachable"
```

We moeten hier echter nog een `reverse` toepassen om te zorgen dat de elementen in hun oorspronkelijke volgorde staan. Het zou makkelijker zijn om de elementen van rechts naar links te overlopen. Dat lukt helaas niet met recursie, omdat de lijstconstructor `:` in Haskell nu eenmaal rechtsassociatief is. We kunnen wel gebruik maken van de `fold` functie-varianten.

Hieronder vind je een recursieve definitie en een voorbeeld van deze functies.

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z                  -- (0)
foldr f z (x:xs) = x `f` foldr f z xs -- (1)
```

```haskell
foldr (+) 0 [1, 2, 3] = 1 + (foldr (+) 0 [2, 3])    -- (1)
                      = 1 + (2 + (foldr (+) 0 [3])) -- (1)
                      = 3 + (foldr (+) 0 [3])
                      = 3 + (3 + (foldr (+) 0 []))  -- (1)
                      = 6 + (foldr (+) 0 [])
                      = 6 + 0                       -- (0)
                      = 6
```

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z                -- (0)
foldl f z (x:xs) = let z' = z `f` x
                   in foldl f z' xs -- (1)
```

```haskell
foldl (+) 0 [1, 2, 3] = foldl (+) (0 + 1) [2, 3]         -- (1)
                      = foldl (+) ((0 + 1) + 2) [3]      -- (1)
                      = foldl (+) (((0 + 1) + 2) + 3) [] -- (1)
                      = ((0 + 1) + 2) + 3                -- (0)
                      = ((1) + 2) + 3
                      = (3) + 3
                      = 6
```

Voorlopig kan je het verschil tussen beiden vereenvoudigd samenvatten:

- `foldr` zal het startelement "rechts van" het laatste element plaatsen, `foldl` zal het startelement "links van" het eerste element plaatsen.
- Bij `foldr` zal (een deel van) de berekening al tussendoor beschikbaar zijn, bij `foldl` pas als de hele lijst overlopen is.

Dit maakt `foldr` bijzonder geschikt voor transformaties van lijsten naar lijsten waarbij de elementen de volgorde moeten behouden, en kan zelfs op oneindige

lijsten werken. `foldl` gebruik je zelden. Daarnaast heb je nog `foldl'` uit `Data.Foldable`, wat een strikte `foldl` is (zie later). Deze gebruik je voor reducties (zoals de som) waarbij de volgorde niet belangrijk is, of je juist de omgekeerde volgorde wilt.

We kunnen `pack` dus best schrijven met behulp van `foldr`.

```
pack' :: Eq a => [a] -> [[a]]
pack' ls = foldr join [] ls -- (met eventuele eta reductie)
  where join l []                        = [l]      : []
        join l ([]:xss)                  = [l]      : xss
        join l ((x:xs):xss) | l == x     = (l:x:xs) : xss
                            | otherwise = [l]      : (x:xs) : xss
```

Als je even went aan de dubbelepunten, is deze notatie een stuk duidelijker. We krijgen een `join` functie, waarin het eerste argument het element is dat we vooraan het tweede argument moeten toevoegen. Om de verschillende mogelijkheden te onderscheiden, kunnen we gewoon dat tweede element deconstrueren.

Een andere recursieve methode die in de les gesuggereerd werd, is ook erg eenvoudig. Helaas zal deze iets trager zijn omdat ze geen staartrecursie toepast (zie later).

```
pack'' ::  Eq a => [a] -> [[a]]
pack'' []                    = []
pack'' [x]                   = [[x]]
pack'' (x:xs) | x == head ys = (x:ys) : yss
              | otherwise    = [x] : ys : yss
  where (ys:yss) = pack'' xs
```

Tenslotte is er ook nog de methode die bestaande functies combineert, die gebruik maakt van de `span` functie (een combinatie van de gesuggereerde `takeWhile` en `dropWhile`).

```
pack''' :: Eq a => [a] -> [[a]]
pack''' [] = []
pack''' l@(x:xs) = eqx : pack''' rest
  where (eqx, rest) = span (== x) l
```

Flatten

De meesten onder ons hadden waarschijnlijk wel door dat het niet de bedoeling was om hier gewoon `concat` in te dienen. Er zijn verschillende andere mogelijkheden, waaronder een recursieve oplossing en één met fold.

```
flatten :: [[a]] -> [a]
flatten []       = []
flatten (xs:xss) = xs ++  flatten xss
-- hmm, lijkt op?  z  `f` foldr   xs
```

```haskell
flatten' :: [[a]] -> [a]
flatten' = foldr (++) []
```

Slice

```haskell
slice :: [a] -> Int -> Int -> [a]
slice l i k = take (k - i) $ drop i l
```

Tails

```haskell
tails :: [a] -> [[a]]
tails [] = [[]]
tails (x:xs) = (x:xs) : tails xs
```

Of met een LC.

```haskell
tails' :: [a] -> [[a]]
tails' l = [ drop x l | x <- [0 .. length l] ]
```

Combinaties

```haskell
combinations :: Int -> [a] -> [[a]]
combinations _ []     = []
combinations 1 xs     = map (:[]) xs
combinations n (x:xs) = map (x:) (combinations (n - 1) xs)
                      ++ combinations n xs
```

We kunnen ook een combinatie van 0 elementen toestaan als we `combinations 0 _ = [[]]` toevoegen en de nu overbodige `combinations 1 xs` weglaten.

Runlength encoding

Met behulp van de eerder gedefinieerde functies `pack` en `flatten`, respectievelijk ingebouwd in Haskell als `group` en `concat` is deze oplossing heel kort.

```haskell
encode :: Eq a => [a] -> [(Int, a)]
encode = map count . group
  where count l = (length l, head l)

decode :: [(Int, a)] -> [a]
decode = concatMap $ uncurry replicate
```

Hier gebruiken we `uncurry :: (a -> b -> c) -> ((a, b) -> c)`, die een functie met twee argumenten omzet in een functie die deze twee argumenten

als tupple aanneemt. `replicate :: Int -> a -> [a]` zal een gegeven element een gegeven aantal keer herhalen.