Design von Algorithmen

Freie Plaetze finden

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Nov 03^{rd}, 2022

Die kleinste freie Zahl

- Gegeben: Liste mit Teilmenge der Zahlen aus $\mathbb{N} \cup \{0\} = \{0, 1, 2, \ldots\}$
- Beispiel $[8, 23, 9, 0, 12]$ oder auch $[5, 3, 2, 1, 0]$
- Problem: Finde die kleinste Zahl, die nicht in der Menge
- "finde den naechsten freien Platz"
- Liste, unsortiert, keine Duplikate

- Konstruiere einen Algorithmus fuer das Problem
- Bestimme seine Effizienz

Schritt 1: naive Loesung

$[5, 3, 2, 1, 0]$

Laufzeit? Warum so geschrieben? Unendlichkeiten

Schritt 1: naive Loesung

$[5, 3, 2, 1, 0]$

```
minFrei xs = head ([0..] \\ xs)
```

Laufzeit? Warum so geschrieben? Unendlichkeiten

Schritt 1: naive Loesung

$[5, 3, 2, 1, 0]$

```haskell
minFrei xs = head ([0..] \\ xs)
```

```haskell
us \\ vs = filter (notElem vs) us

notElem [] r = True
notElem (l:ls) r = l /= r && notElem ls r
```

Laufzeit? Warum so geschrieben? Unendlichkeiten

Grundueberlegung zur Verbesserung

- $[5, 3, 2, 1, 0]$, Laenge: 5
- $[0, 1, 2, 3, 4, 5, 6]$, Laenge: 7

Grundueberlegung zur Verbesserung

- $[5, 3, 2, 1, 0]$, Laenge: 5
- $[0, 1, 2, 3, 4, 5, 6]$, Laenge: 7

Gegeben $|xs| = n$, muss es in $[0, \dots, n]$ min. eine Zahl geben die nicht in xs ist
Warum?

Grundueberlegung zur Verbesserung

- $[5, 3, 2, 1, 0]$, Laenge: 5
- $[0, 1, 2, 3, 4, 5, 6]$, Laenge: 7

Gegeben $|xs| = n$, muss es in $[0, \ldots, n]$ min. eine Zahl geben die nicht in xs ist
Warum?

Die kleinste Zahl nicht in xs erfuellt ist auch nicht in:
$\{x | x \leftarrow xs, x \leq n\}$

```
filter (<=n) xs
```

Array Algorithmus

Konstruiere ein Array mit $n+1$ Elementen zwischen 0 und n, initial alle `False`. Setze die xs auf `True` und frage dann:

```
search :: Array Int Bool -> Int
search = length . takeWhile id . elems



takeWhile id == takeWhile (\x -> x==True)
  (\x -> x==True) == (==True) == id

takeWhile f [] = []
takeWhile f (x:xs) = if f x then x : takeWhile f xs else []
```

Laufzeit?

Array-Konstruktion

magic ahead!

```
minFreiA = search . checklist

checklist :: [Int] -> Array Int Bool
checklist xs = accumArray (||) False (0,n)
  (zip (filter (<=n) xs) (repeat True))
  where n = length xs


accumArray
  :: Ix i => (e -> a -> e) -> e
  -> (i, i) -> [(i, a)] -> Array i e
```

Ix? assoziative Listen? Array magic? Laufzeit?

Array-Konstruktion mittels ST

```
checklistST :: [Int] -> Array Int Bool
checklistST xs = runSTArray (do
  let n = length xs
  a <- newArray (0,n) False
  sequence_ [ writeArray a x True
            | x <- xs, x <= n ]
  return a)
```

Typen der wichtigen Funktionen? Weshalb ist hier das “magische” ST wichtig?

minVon (generisch)

Es passiert nicht viel, `minVon 0 == minFrei` ist die gesuchte Funktion

```
minVon :: Int -> [Int] -> Int
minVon a xs = head ([a..] \\ xs)
```

minVon (generisch)

Es passiert nicht viel, `minVon 0 == minFrei` ist die gesuchte Funktion

```
minVon :: Int -> [Int] -> Int
minVon a xs = head ([a..] \\ xs)
```

```
minFrei (xs ++ ys) == minFrei xs `something` minFrei ys
```

Was muss gelten?

Law and Order

Law and Order

```
(as++bs) \\ cs == (as \\ cs) ++ (bs \\ cs)
as \\ (bs++cs) == (as \\ bs) \\ cs
(as \\ bs) \\ cs == (as \\ cs) \\ bs
```

cf. Mengen!

Law and Order

```
(as++bs) \\ cs == (as \\ cs) ++ (bs \\ cs)
as \\ (bs++cs) == (as \\ bs) \\ cs
(as \\ bs) \\ cs == (as \\ cs) \\ bs
```

cf. Mengen!

Sei nun `as \\ us == as` und `bs \\ us == bs` (die Listen also jeweils disjoint)

Law and Order

```
(as++bs) \\ cs == (as \\ cs) ++ (bs \\ cs)
as \\ (bs++cs) == (as \\ bs) \\ cs
(as \\ bs) \\ cs == (as \\ cs) \\ bs
```

cf. Mengen!

Sei nun `as \\ us == as` und `bs \\ us == bs` (die Listen also jeweils disjoint)

dann gilt:

```
(as++bs) \\ (us++vs) == (as \\ us) ++ (bs \\ vs)
```

Erstelle folgende Teillisten

```
as = [0..b-1]       -- endlich
bs = [b..]          -- unendlich
us = filter (<b) xs   -- partition von xs
vs = filter (>=b) xs  -- dito
```

Erstelle folgende Teillisten

```
as = [0..b-1]          -- endlich
bs = [b..]             -- unendlich
us = filter (<b) xs     -- partition von xs
vs = filter (>=b) xs    -- dito
```

```
[0..] \\ xs = (as \\ us) ++ (bs \\ vs)
  where (us,vs) = partition (<b) xs
```

Erstelle folgende Teillisten

```
as = [0..b-1]         -- endlich
bs = [b..]            -- unendlich
us = filter (<b) xs    -- partition von xs
vs = filter (>=b) xs   -- dito
```

```
[0..] \\ xs = (as \\ us) ++ (bs \\ vs)
  where (us,vs) = partition (<b) xs
```

```
partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = foldr (select p) ([],[]) xs

select :: (a -> Bool) -> a -> ([a],[a]) -> ([a],[a])
select p x ~(ts,fs)
  | p x = (x:ts, fs)
  | otherwise = (ts,x:fs)
```

Was folgt?

```
head (xs++ys) = if null xs then head ys else head xs
```

Was folgt?

```
head (xs++ys) = if null xs then head ys else head xs
```

```
minFrei xs ==
  if null ([0..b-1] \\ us)
  then head ([b..] \\ vs)
  else head ([0..b-1] \\ us)
  where (us,vs) = partition (<b) xs
```

Komplexitaet von `null ([0..b-1] \\ us)` ?

Was folgt?

```
head (xs++ys) = if null xs then head ys else head xs
```

```
minFrei xs ==
  if null ([0..b-1] \\ us)
  then head ([b..] \\ vs)
  else head ([0..b-1] \\ us)
  where (us,vs) = partition (<b) xs
```

Komplexitaet von `null ([0..b-1] \\ us)` ?

```
null ([0..b-1] \\ us) === length us == b
```

Divide and Conquer

Divide and Conquer

```haskell
minFreiDC = minVonDC 0

minVonDC :: Int -> [Int] -> Int
minVonDC a xs
  | null xs = a
  | length us == b-a = minVonDC b vs
  | otherwise = minVonDC a us
  where
    (us,vs) = partition (<b) xs
    b = ?
    n = length xs
```

Divide and Conquer

```
minFreiDC = minVonDC 0

minVonDC :: Int -> [Int] -> Int
minVonDC a xs
  | null xs = a
  | length us == b-a = minVonDC b vs
  | otherwise = minVonDC a us
  where
    (us,vs) = partition (<b) xs
    b = ?
    n = length xs
```

```
    b = a + 1 + n `div` 2
```

minimiert das Maximum der Laenge der beiden Listen *us*, *vs*

Asymptotik

- Falls $n \neq 0$ und $|us| < b - a$:
  $|us| \leq n \text{ div } 2 < n$
- $|us| = b - a$:
  $|vs| = n - n \text{ div } 2 - 1 \leq n \text{ div } 2$
- $T(n) = T(n \text{ div } 2) + \Theta(n)$
- womit: $T(n) = \Theta(n)$

Ausblick: Monaden

Was ist dieses ST? Are you kidding me?

```
newtype ST s a = ST (STRep s a)
type STRep s a = State# s -> (# State# s, a #)

instance Functor (ST s) where
    fmap f (ST m) = ST $ \ s ->
      case (m s) of { (# new_s, r #) ->
      (# new_s, f r #) }

instance Monad (ST s) where
    (ST m) >>= k
      = ST (\ s ->
        case (m s) of { (# new_s, r #) ->
        case (k r) of { ST k2 ->
        (k2 new_s) }})
```

Ausblick: Monaden

Was ist dieses ST? Are you kidding me?

```
type ST s a = s -> (s, a)

instance Functor (ST s) where
    fmap f m = \ s ->
      case (m s) of { (new_s, r) ->
      (new_s, f r ) }

instance Monad (ST s) where
    m >>= k
      = \ s ->
        case (m s) of { (new_s, r) ->
        case (k r) of { k2 ->
        (k2 new_s) }}
```

Einfache Funktionskombinatoren

```
sum (map (*2) (filter even [1..100]))

sum . map (*2) $ filter even [1..100]

:type sum . map (*2) . filter even -- ??

infixr 9 .
(.)     :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)

infixr 0 $
($) :: (a -> b) -> a -> b
f $ x =  f x
```

Kompliziertere Listengeneratoren

```
[(x,y) | x <- [1..10], y <- [x..10]]
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
ccteven xss = [x | xs <- xss, even (length xs)
                 , x <- xs, even x]

faktoren n = [x|x <- [1..n], n `mod` x == 0]
prime n = faktoren n == [1,n]
primes = [x | x <- [2..], prime x]
-- take 100 primes
```

Der Reisverschluss

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f = go
  where
    go [] _ = []
    go _ [] = []
    go (x:xs) (y:ys) = f x y : go xs ys

zip = zipWith (,) -- testen!
```

Mehr Rekursion auf Listen

```
filter :: (a->Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs) = if f x then x : filter f xs else filter f

map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys   -- Achtung: infixr 5 : ++

reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

reverse' xs = let
    rev [] acc = acc
    rev (x:xs) acc = rev xs (x:acc)
  in go xs []
```

Hausaufgabe

```
data Ausdruck
  = Wert Int
  | Ausdruck :+ Ausdruck
  | Ausdruck :* Ausdruck
  deriving (Show)

instance Num (Ausdruck) where
  fromInteger x = Wert x     -- oh dear

ausdruck = Wert 1 :+ (Wert 2 :* Wert 3) -- 1 :+ (2 :* 3)

tiefe :: Ausdruck -> Int
tiefe = error "Tiefe␣des␣Ausdrucksbaumes"
wert :: Ausdruck -> Int
wert = error "Wert␣dieses␣Ausdruck"
```

Binaerbaeume

```
data Tree a = Tip | Node (Tree a) a (Tree a)

empty = Tip :: Tree a

insert Tip a = Node Tip a Tip
insert (Node ls x rs) a
  | a == x = Node ls x rs
  | a < x = Node (insert ls a) x rs
  | a > x = Node ls x (insert rs a)

exists Tip a = False
exists (Nodes ls x rs) a
  | a==x = True
  | a< x = exists ls a
  | a> x = exists rs a

-- data Tree k v = ...
```

Deep Magic!

```
data V (k :: Nat) a where
  Nul :: V 0
  (:>) :: a -> V k -> V (k+1)

length :: forall k . V (k::Nat) -> Int
length _ = fromIntegral (natVal Proxy :: Proxy k)
```

Datentypen

- Neue Datentypen `data TyCon a b = DataCon a b`
- TypAliase `type T = S`
- `newtype TyCon a = DataCon a`

```haskell
data TyData a b c = HereD a b | ThereD c

type MyInt = Int

newtype NewTy X = NewTy Int
```

The Roller Coaster

```haskell
data Person = P { age :: Int, height :: Int }
guard :: Person -> Bool
guard (P a h) = not (a < 12 || h < 100)
coaster = map guard

guys =
  coaster [P 11 110, P 8 90, P 117 13, P 13 130]

newtype Age = Age Int
newtype Height = Height Int
... P { age :: Age, height :: Height }
```

Der Guard lässt nur das dritte und vierte Kind fahren

Haskell Prelude

```haskell
head [1..5] == 1
tail [1..5] == [2..5]
[1..5] !! 2 == 3
take 2 [1..5] == [1..2]
drop 2 [1..5] == [3..5]
length [1..5] == 5
take 2 [1..10^100] == [1..2]
take 2 [1..] == [1..2]
sum [] == 0
sum [1..5] == 15
product [] == 1
product [1..5] == 120
reverse [1..5] == [5,4..1]
```

Implementation nach `data List`