Tower of Hanoi

April 23, 2022

Recursion, getting started

Factorial

```
factorial :: Int -> Int
factorial 0 = 1
factorial x = x * factorial (x - 1)
```

Factorial

```
  factorial 3
= 3 * factorial 2
= 3 * 2 * factorial 1
= 3 * 2 * 1 * factorial 0
= 3 * 2 * 1 * 1
= 6
```

Fibonacci sequence

1 1 2 3 5 8 13 21 ...

Fibonacci sequence

1 1 2 3 5 8 13 21 ...

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

Fibonacci sequence

1 1 2 3 5 8 13 21 ...

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

```
  fib 4
= fib 3           + fib 2
= fib 2 + fib 1   + 1
= 1     + 1       + 1
= 3
```

Lists

Lists

```
x = [1, 2, 3] :: [Int]
y = [4, 5, 6] :: [Int]

hi = ['h', 'i'] :: [Char]
hi = "hi" :: String
-- String == [Char]

booleans = [True, False] :: [Bool]
```

Lists

Simple operations on lists

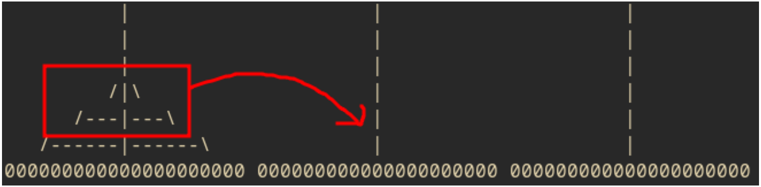
```
x = [1, 2, 3] :: [Int]
z = 0 : x = [0, 1, 2, 3]

head x == 1
tail x == [2, 3]
x ++ y == x <> y == [1, 2, 3, 4, 5, 6]
```

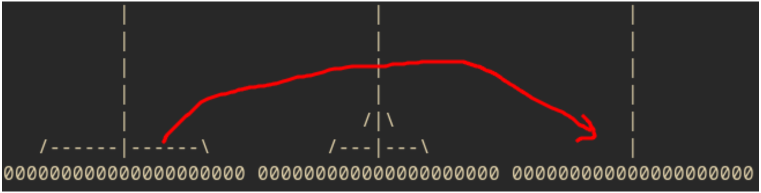
Solving Tower of Hanoi with recursion

Idea

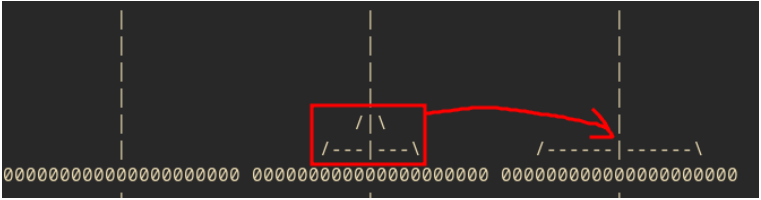
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Tower of Hanoi

```
          |                     |                     |
          |                     |                     |
          |                     |                     |
          |                     |                    /|\
          |                     |                 /---|---\
          |                     |              /------|------\
000000000000000000000 000000000000000000000 000000000000000000000
```

- Move all of the discs except the last one from peg a to peg b
- Move the last disc from peg a to peg c
- Move all of the discs on peg b to peg c

Tower of Hanoi

```
data Peg = PegA | PegB | PegC
  deriving (Show)
type Move = (Peg, Peg)
```

Tower of Hanoi

```haskell
data Peg = PegA | PegB | PegC
  deriving (Show)
type Move = (Peg, Peg)

solve
  :: Int -- number of discs
  -> Peg -- from
  -> Peg
  -> Peg -- to
  -> [Move]
```

Tower of Hanoi

```haskell
-- target: move all discs from A to C
solve 1 x y z = [(x, z)]
solve n x y z =
  -- Move all of the discs except the
  -- last one from peg a to peg b
  solve (n - 1) x z y
  -- Move the last disc
  -- from peg a to peg c
  ++ [(x, z)]
  -- Move all of the
  -- discs on peg b to peg c
  ++ solve (n - 1) y x z
```

Tower of Hanoi

```
solve 1 x y z = [(x, z)]
solve n x y z =
  solve (n - 1) x z y
  ++ [(x, z)]
  ++ solve (n - 1) y x z
```

```
   solve 3 a b c
= solve 2 a c b
++ [(PegA, PegC)]
++ solve 2 b a c
= solve 1 a b c ++ [(PegA, PegB)] ++ solve 1 c a b
++ [(PegA, PegC)]
++ solve 1 b c a ++ [(PegB, PegC)] ++ solve 1 a b c
= [(PegA, PegC), (PegA, PegB), [(PegC, PegB)]
++ [(PegA, PegC)]
++ [(PegA, PegC), (PegA, PegB), (PegC, PegB)]
```

More Lists

More lists

```
map (\x -> x + 1) [1, 2, 3]
```

[	1	2	3	]
map (+1)	$\downarrow$	$\downarrow$	$\downarrow$	map (+1)
[	2	3	4	]

More lists

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

```haskell
x = [1, 2, 3]
x = (1:[2, 3])
```

More lists

```haskell
replicate :: Int -> a -> [a]
replicate 5 'x' = ['x', 'x', 'x', 'x', 'x'] = "xxxxx"

-- String = [Char]
```

More lists

```haskell
replicate :: Int -> a -> [a]
replicate :: Int -> Char -> [Char]
replicate 0 _ = []
replicate n c = c : replicate (n - 1) c

-- String = [Char]
```

More lists

```
filter (\x -> x > 3) [5, 4, 3, 6, 7, 8]
= [5,4,6,7,8]
filter odd [5, 4, 3, 6, 7, 8]
= [5,3,7]
```

More lists

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

More lists

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

More lists

```haskell
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) =
  let small = quicksort (filter (\a -> a <= x) xs)
      big = quicksort (filter (\a -> a > x) xs)
  in  small ++ [x] ++ big
```

More lists

```haskell
isPrime :: Int -> Int -> Bool
isPrime d n =
  if d >= n - 1
    then True
    else ((mod n d) /= 0) && (isPrime (d + 1) n)

primesUnder100 = filter (\x -> isPrime 2 x) [1..100]
```

More lists

Goldbach conjecture

```
sums = [x + y | x <- primesUnder100, y <- primesUnder100]
2 `elem` sums
4 `elem` sums
6 `elem` sums
```

More lists

```
intercalate ", " ["line one", "line two", "line three"]
= "line one, line two, line three"
```

More lists

Try implementing *intercalate* by yourself!

Drawing the console output

Drawing the console output

```
drawTower :: [Int] -> Int -> Int -> [String]
drawTower discs towerHeight baseLen = ...
*Toh> drawTower [3, 2, 1] 5 15
[ "       |       "
, "       |       "
, "   /---|---\\   "
, "  /----|----\\  "
, " /-----|-----\\ "
, "000000000000000"
]
```

Drawing the console output


```haskell
drawBase
  = replicate baseLen '0'
```


```haskell
drawBar = replicate
            (baseLen `div` 2)
        <> "|"
        <> replicate
            (baseLen `div` 2)
```

Drawing the console output

```
   /---|---\
```

```haskell
discSize i = (baseLen - 2 * i - 2) `div` 2
-- i is the amount of whitespace at the beginning
drawDisc i = replicate i ' ' <> "/"
          <> replicate (discSize i) '-'
          <> "|"
          <> replicate (discSize i) '-'
          <> "\\" <> replicate i ' '
```

Putting them together

Drawing the console output

```haskell
drawTower :: [Int] -> Int -> Int -> [String]
drawTower
  = replicate (towerHeight - length discs) drawBar
  ++ (map drawDisc discs)
  ++ [drawBase]
```

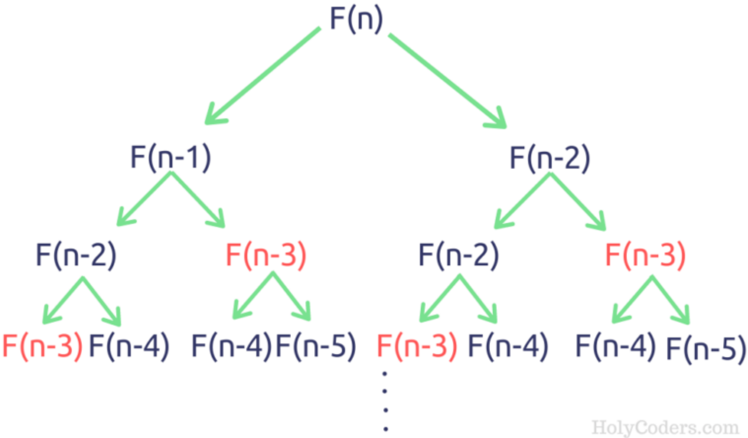
```
*Toh> drawTower [3, 2, 1] 5 15
```

```
       |
       |
```

```
    /---|---\
   /----|----\
  /-----|-----\
```

```
000000000000000
```

Fibonacci sequence



Source:
`https://holycoders.com/algorithms-fibonacci-sequence/`

Fibonacci sequence

```
fibHelper :: Int -> Int -> Int -> Int
fibHelper x _ 1 = x
fibHelper pp p n = fibHelper p (p + pp) (n - 1)
```

Fibonacci sequence

```
fibHelper :: Int -> Int -> Int -> Int
fibHelper x _ 1 = x
fibHelper pp p n = fibHelper p (p + pp) (n - 1)

fib :: Int -> Int
fib n = fibHelper 1 1 n
```

Fibonacci sequence

```
fibHelper :: Int -> Int -> Int -> Int
fibHelper x _ 1 = x
fibHelper pp p n = fibHelper p (p + pp) (n - 1)
```

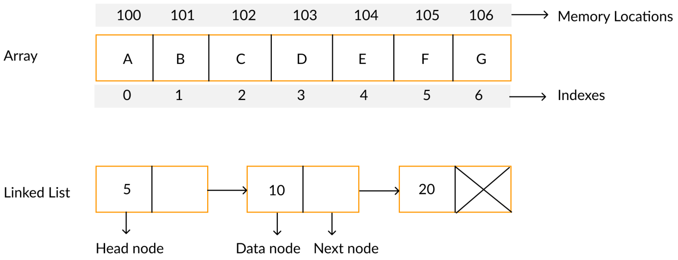
```
  fib 4
= fibHelper 1 1 4
= fibHelper 1 (1 + 1) 3
= fibHelper 1 2       3
= fibHelper 2 (2 + 1) 2
= fibHelper 2 3       2
= fibHelper 3 5       1
= 3
```

Lists

- Lists in haskell are stored as linked lists

Lists

- Lists in haskell are stored as linked lists

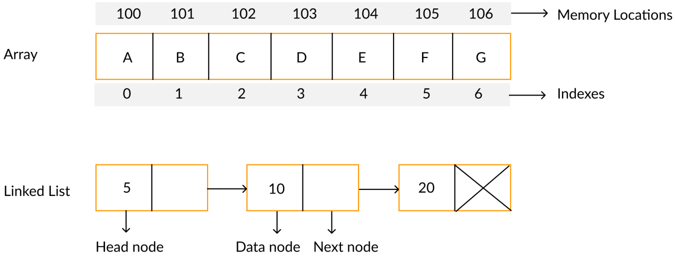


Source: `https://www.faceprep.in/data-structures/linked-list-vs-array/`

- Random access is slow, use Vector for random access (e.g. store the no of discs on each peg)

Lists

- Lists in haskell are stored as linked lists



Source: `https://www.faceprep.in/data-structures/linked-list-vs-array/`

- Random access is slow, use Vector for random access (e.g. store the no of discs on each peg)
  - They are the "control structure" of haskell