50001 - Algorithm Analysis and Design - Lecture 14

Oliver Killane

28/11/21

Lecture Recording

Lecture recording is available here

Randomized Algorithms

An algorithm that uses random values to produce a result.

Algorithm Type	Running time	Correct Result
Monte Carlo	Predicatable	Unpredictably
Las Vegas	Unpredictable	Predictably

Random Generation

Functions are deterministic (always map same inputs to same outputs), this is known as **Leibniz's law** or the **Law of indiscernibles**:

$$x = y \Rightarrow f x = f y$$

We can exhibit pesudo random behaviour using an input that varies

explicitly	(e.g Random numbers through seeds)
implicitly	(e.g Microphone or camera noise)

Inside IO Monad

We can use basic random through the IO monad like this:

```
import Control.Monad.Random (getRandom)

main :: IO ()
main = do
    x <- getRandom :: IO Int
    print (42 + x)
```

However using the **IO monad** is too specific, we may want to use random numbers in other contexts.

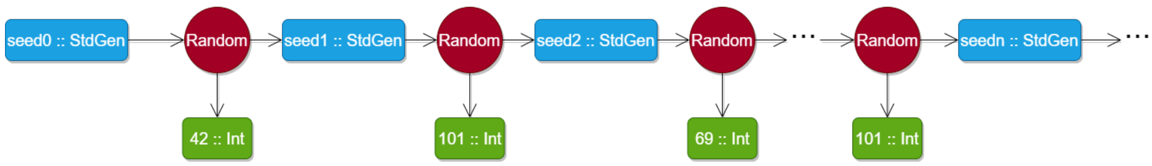
StdGen

In haskell we can use **Stdgen**.

```
import System.Random (StdGen)

-- Create a source of randomness from an integer seed
mkStdGen :: Int -> StdGen

-- Generate a random interger, and a new source of randomness
random :: StdGen -> (Int, StdGen)

-- Generate an infinite list of random numbers using an initial seed
-- (source of random).
randoms :: StdGen -> [Int]
randoms seed = x:randoms seed' where (x, seed') = random seed

-- In order to generate random value for any type, a typeclass is used
class Random a where
```

```haskell
random   :: StdGen -> (a, StdGen)
randoms  :: StdGen -> [a]

-- Random over a (R)ange
randomR  :: (a,a)  -> StdGen -> (a, StdGen)
randomRs :: (a,a)  -> StdGen -> [a]
```

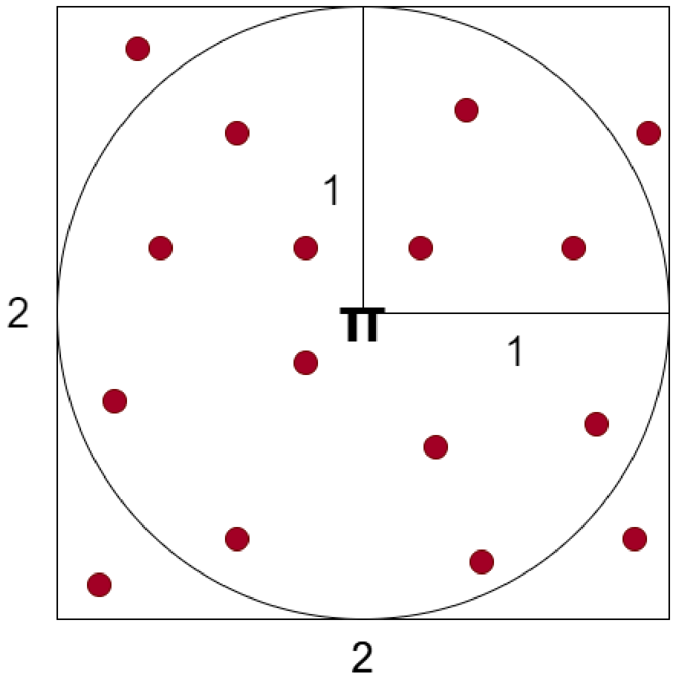
By passing the newly generated **StdGen** we can generate new values based on the original seed.



With Random Monad

Rather than passing **StdGen** seeds through the program, we can use the **MonadRandom** monad which internally uses this value.

Randomized π



(Monte Carlo Algorithm - known number of samples, known running time per sample) To estimate π, find the proportion of randomly selected spots that are within the circle.

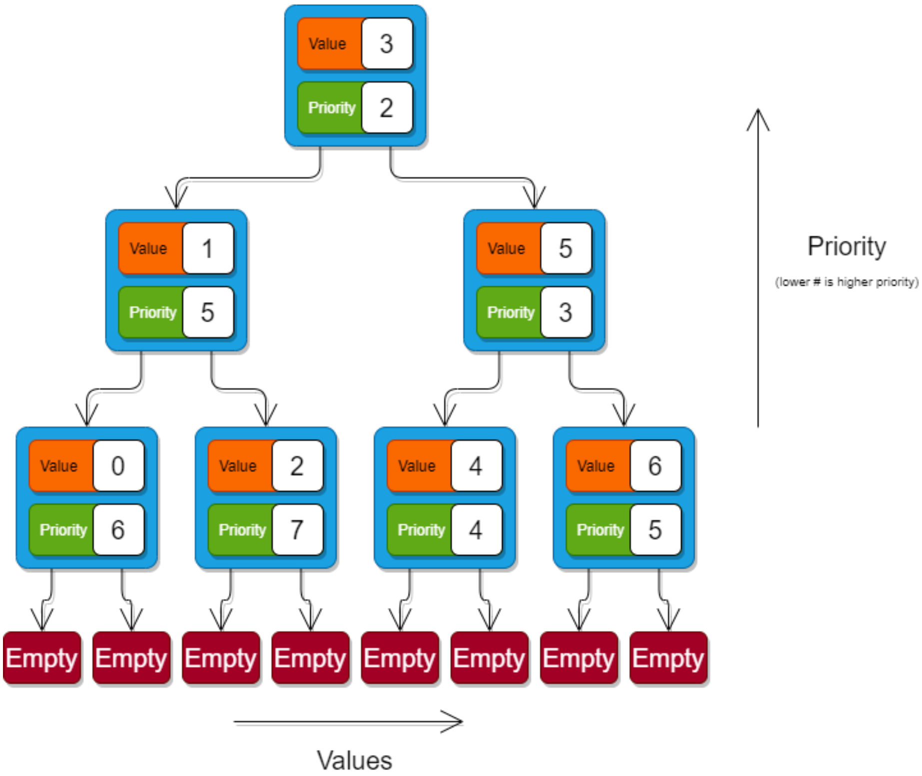
Area of square	$2 \times 2 = 4$
Area of circle	$\pi \times 1^2 = \pi$
Probability in circle	$\frac{\pi}{4}$

Once we have the proportion, we can multiply by 4 to get an estimate of π.

```haskell
import Control.Monad.Random (getRandomR, randomRs, MonadRandom)
import System.Random (mkStdGen, StdGen)

-- Here we can use one quarter of the circle, hence if the distance from the
-- bottom left (0,0) to the point is within 1 then it is in the circle.
inside :: Double -> Double -> Bool
inside x y = 1 >= x * x + y * y

-- Take 1000 samples and return 4 * the proportion.
montePi :: MonadRandom m => m Double
montePi = loop samples 0
  where
    samples = 10000
    loop 0 m = return (4 * fromIntegral m / fromIntegral samples)
    loop n m = do
      x <- getRandomR (0,1)
      y <- getRandomR (0,1)
      loop (n-1) (if inside x y then m + 1 else m)




-- Using a stream of random numbers (RandomRs)

-- Get pairs of random numbers from the stream
pairs :: [a] -> [(a,a)]
pairs (x:y:ls) = (x,y):pairs ls

-- From the pairs of random numbers, get the proportion of points inside the
-- circle and use to get pi.
montePi' :: Double
montePi' = 4 * hits src / fromIntegral samples
  where
    samples = 10000
    hits    = fromIntegral .
              length .
              filter (uncurry inside) .
              take samples .
              pairs
    src     = randomRs (0, 1) (mkStdGen 42) :: [Double]
```

Treaps

Simultaneously a **Tree** and a **Heap**. Stores values in order, while promoting higher priority nodes to the top of the tree.




```haskell
-- Node contains child treaps, as well as value (a) and the priority (Int)
data Treap a = Empty | Node (Treap a) a Int (Treap a)

-- Normal tree search using values
member :: Ord a => a -> Treap a -> Bool
member x (Node l y _ r)
  | x == y      = True
  | x < y       = member x l
  | otherwise = member x r
member _ Empty = False

-- Priority based insert
pinsert :: Ord a => a -> Int -> Treap a -> Treap a
pinsert x p Empty = Node Empty x p Empty
pinsert x p t@(Node l y q r)
  | x == y      = t
  | x < y       = lnode (pinsert x p l) y q r
  | otherwise = rnode l y q (pinsert x p r)

-- rotate right (check left node)
lnode :: Treap a -> a -> Int -> Treap a -> Treap a
lnode Empty y q r = Node Empty y q r
lnode l@(Node a x p b) y q c
  | q > p       = Node a x p (Node b y q c)
  | otherwise = Node l y q c

-- rotate left (check right node)
rnode :: Treap a -> a -> Int -> Treap a -> Treap a
rnode l y q Empty = Node l y q Empty
```

```haskell
rnode a x p r@(Node b y q c)
  | q < p     = Node (Node a x p b) y q c
  | otherwise = Node a x p r

-- delete node by recursively searching, then delete and merge subtrees
delete :: Ord a => a -> Treap a -> Treap a
delete x Empty = Empty
delete x (Node a y q b)
  | x == y    = merge a b
  | x < y     = Node (delete x a) y q b
  | otherwise = Node a y q (delete x b)

merge :: Treap a -> Treap a -> Treap a
merge Empty r = r
merge l Empty = l
merge l@(Node a x p b) r@(Node c y q d)
  | p < q     = Node a x p (merge b r)
  | otherwise = Node (merge l c) y q d
```