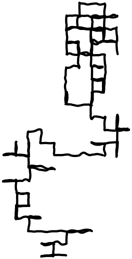
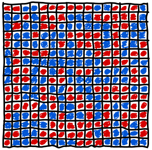


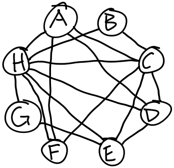
Purely Random

5 3 E 5 1 2 8 C
E 8 8 C 1 7 3 8
5 4 6 8 B D 4 C
6 8 4 3 0 6 5 7
5 D 2 C 8 C 7 C
4 9 0 F 6 8 4 9
6 A E 2 E 6 9 7
5 8 5 6 C E 9 9

Gary Fredericks

Clojure/West 2015





java.util.Random

```
user> (def r (java.util.Random. 42))
#’user/r
user> (.nextInt r)
-1170105035
user> (.nextInt r)
234785527
```

java.util.Random in Clojure

```
(defn create
  [seed]
  {:state (bit-xor seed 0x5deece66d)})

(defn next-int
  [{:keys [^long state]}]
  (let [new-state (-> state
                      (unchecked-multiply 0x5deece66d)
                      (unchecked-add 0xb))
        x (-> new-state
              (bit-shift-right 16)
              (unchecked-int))]
    [x {:state new-state}]))
```

Mutable vs. Immutable

```
user> (def r (java.util.Random. 42))
#’user/r
user> (.nextInt r)
-1170105035
user> (.nextInt r)
234785527

;; immutable version
user> (def r (create 42))
#’user/r
user> r
{:state 25214903879}
user> (next-int r)
[-1170105035 {:state 8602080079250839110}]
user> (next-int r)
[-1170105035 {:state 8602080079250839110}]
user> (next-int (second *1))
[234785527 {:state 7522434139496587225}]
```

Roadmap

- Splittability and Composition
  - Basic Example, Definitions
  - Case Study: test.check
- Implementing Splittable RNGs in Clojure
  - Poorly
  - Better
  - Faster

Splittability and Composition

Splittability and Composition

A Tale of Two Seqs

Requirements

```clojure
(defn pair-of-lazy-seqs
  "Given a seed, returns [xs ys]
  where xs and ys are both
  (different) lazy infinite seqs
  of random numbers."
  [seed]
  ;; ???
  )
```

With `java.util.Random`

```clojure
(defn pair-of-lazy-seqs
  [seed]
  (let [r (java.util.Random. seed)]
    [(repeatedly #(.nextInt r))
     (repeatedly #(.nextInt r))]))
```

Let's use it

```
(let [[xs ys] (pair-of-lazy-seqs 42)]
  [(take 4 xs) (take 4 ys)])
=>
[(-1170105035 234785527 -1360544799 205897768)
 (1325939940 -248792245 1190043011 -1255373459)]
```

Let's use it

```
(let [[xs ys] (pair-of-lazy-seqs 42)]
  [(take 4 xs) (take 4 ys)])
=>
[(-1170105035 234785527 -1360544799 205897768)
 (1325939940 -248792245 1190043011 -1255373459)]

(let [[xs ys] (pair-of-lazy-seqs 42)]
  [(first xs) (first ys)])
 => [-1170105035 234785527]
```

With `java.util.Random`

```
(defn pair-of-lazy-seqs
  [seed]
  (let [r (java.util.Random. seed)]
    [(repeatedly #(.nextInt r))
     (repeatedly #(.nextInt r))]))
```

With the immutable clojure RNG

```clojure
(defn random-nums
  [rng]
  (lazy-seq
   (let [[x rng2] (next-int rng)]
     (cons x (random-nums rng2)))))

(defn pair-of-lazy-seqs
  [seed]
  (let [rng (create seed)]
    [(random-nums rng)
     (random-nums ; ????
      )]))
```

Concept Space

	Linear	Splittable
Mutable	.nextInt ⇒ x	.nextInt ⇒ x .split ⇒ rng2
Immutable	next-int ⇒ [x rng2]	rand-int ⇒ x split ⇒ [rng1 rng2]

With a splittable RNG

```clojure
(defn random-nums
  [rng]
  (lazy-seq
   (let [[rng1 rng2] (split rng)
         x (rand-int rng1)]
     (cons x (random-nums rng2)))))

(defn pair-of-lazy-seqs
  [seed]
  (let [rng (create seed)
        [rng1 rng2] (split rng)]
    [(random-nums rng1)
     (random-nums rng2)]))
```

Splittability and Composition

test.check

gen-xs-and-x

```
(def gen-xs-and-x
  "Generates a pair [xs x] where xs is a list of
  numbers and x is a number in that list."
  (gen/bind (gen/not-empty (gen/list gen/nat))
            (fn [xs]
              (gen/tuple (gen/return xs)
                         (gen/elements xs)))))

(gen/sample gen-xs-and-x)
=>
([(0) 0]
 [(3 3 0) 0]
 [(1 2) 2]
 [(2 0 3 1) 1]
 [(4 0 1 3) 1]
 ...)
```

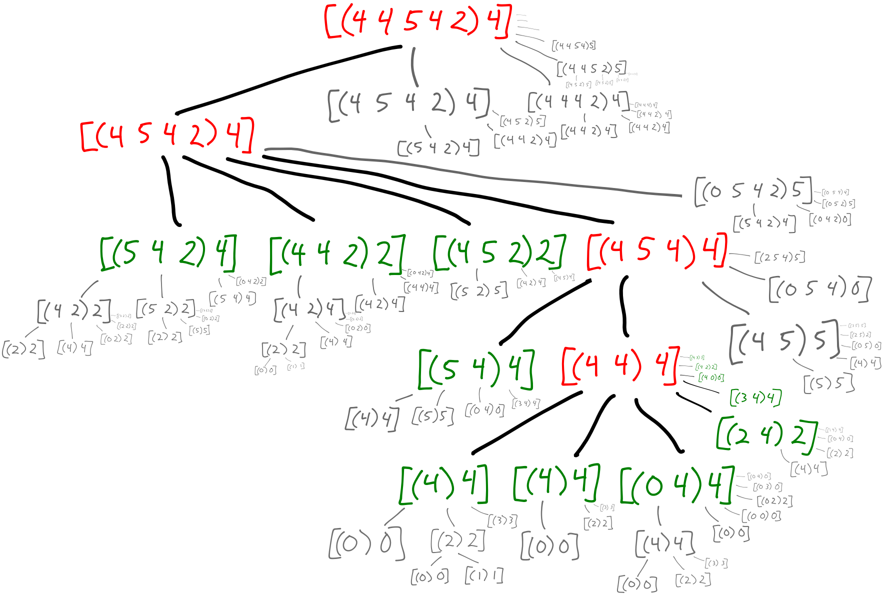
lists-don’t-have-duplicates

```clojure
(def lists-don’t-have-duplicates
  (prop/for-all [[xs x] gen-xs-and-x]
    (let [x-count (->> xs
                       (filter #{x})
                       (count))]
      (= 1 x-count))))
```

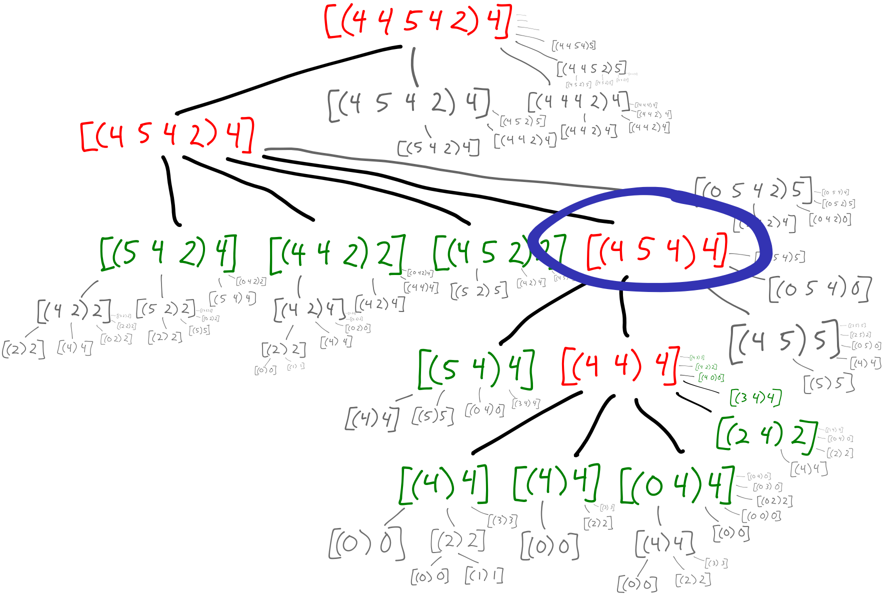
test.check shrinking

```
user> (quick-check 100 lists-don’t-have-duplicates)
{:fail [[(4 4 5 4 2) 4]],
 :failing-size 6,
 :num-tests 7,
 :result false,
 :seed 1426989885725,
 :shrunk {:depth 3,
          :result false,
          :smallest [[(4 4) 4]],
          :total-nodes-visited 16}}
```

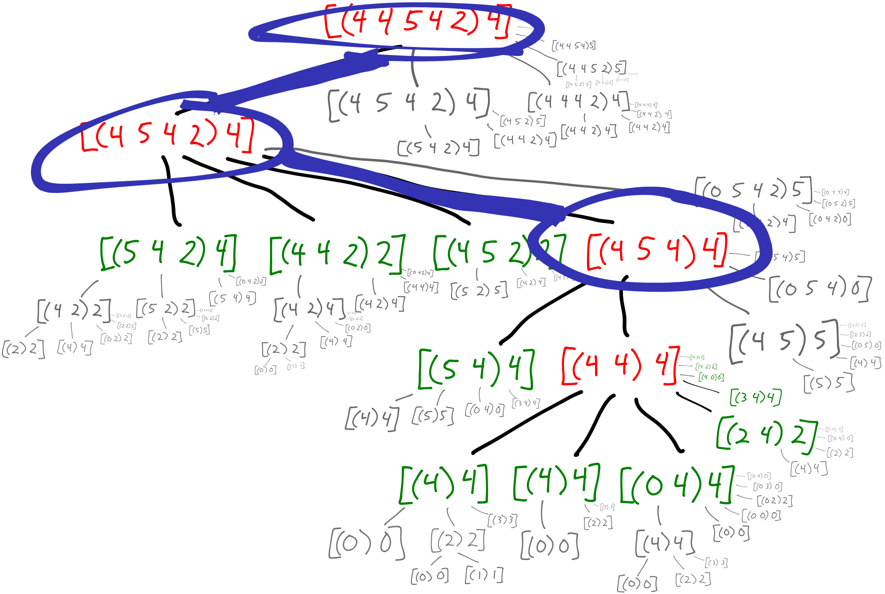
test.check shrink tree



test.check shrink tree



test.check shrink tree



test.check

The Problem

- the lazy shrink-tree is nondeterministic

The Solution

- Use an immutable, splittable RNG.
  - But where do you find such a thing?

Splittability and Composition

Summary

Okay So

- Linear RNGs hinder composition
  - Programs are either nondeterministic or impossible to write
- Splittable RNGs are less common, but composition-friendly
- test.check impl is fragile because of its linear RNG

Implementations

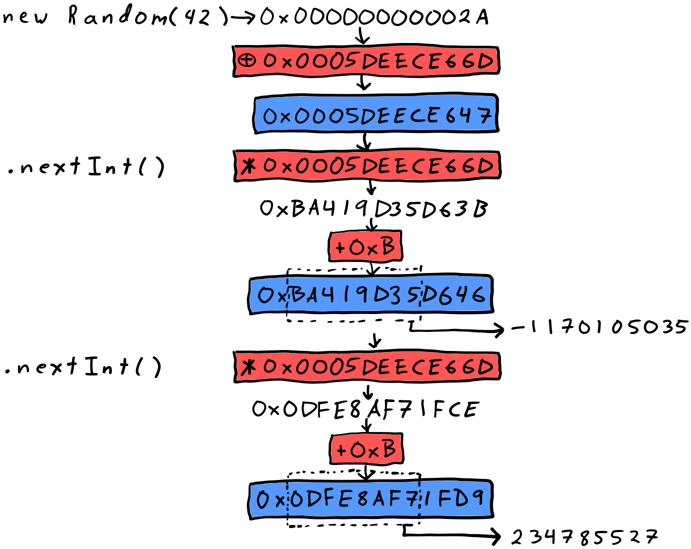
Implementing Splittable RNGs in Clojure

- Poorly
- Better
- Faster

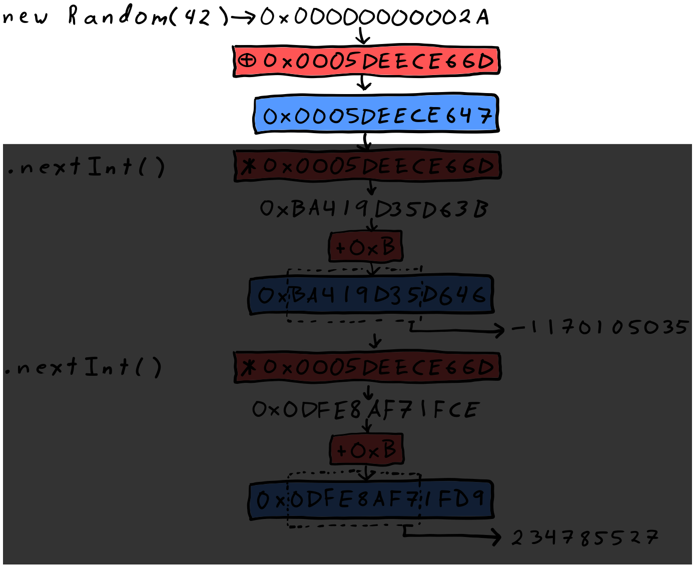
Implementations

Low Quality Implementations

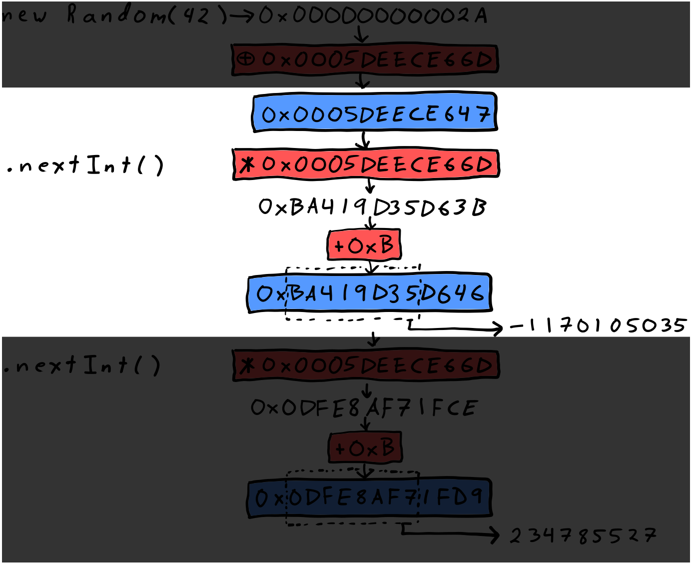
java.util.Random



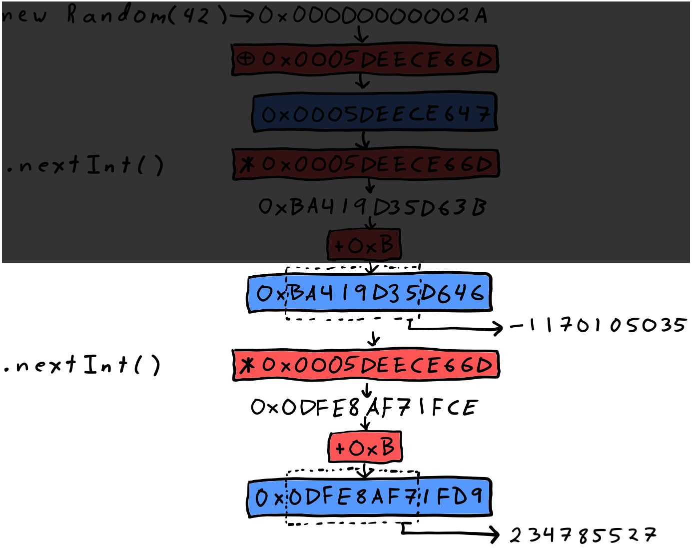
java.util.Random



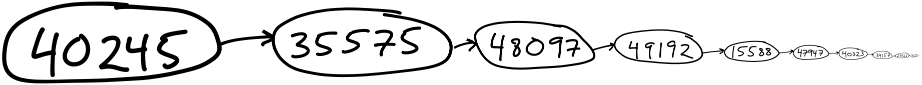
java.util.Random



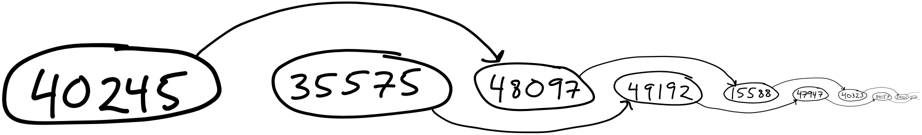
java.util.Random



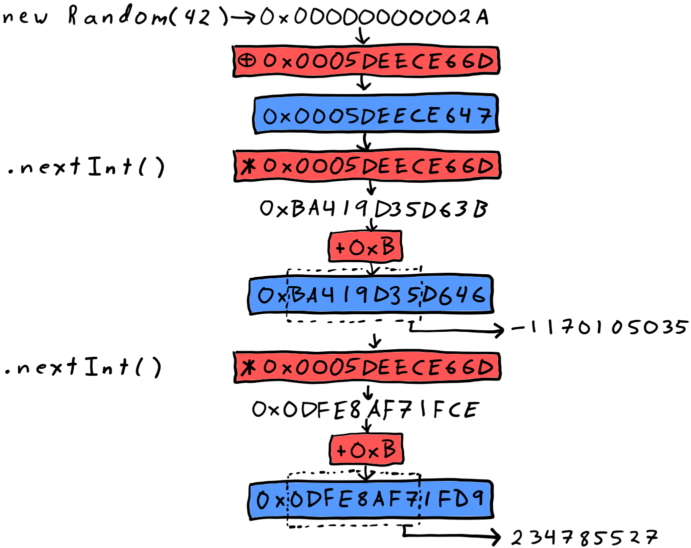
`java.util.Random` as a lazy seq



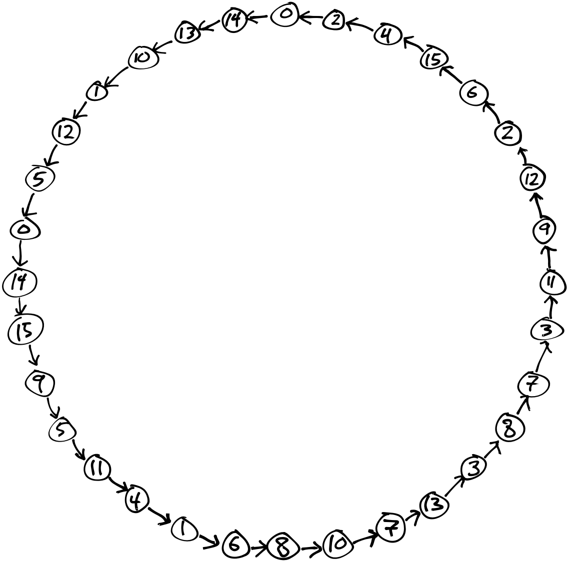
java.util.Random: splitting the seq



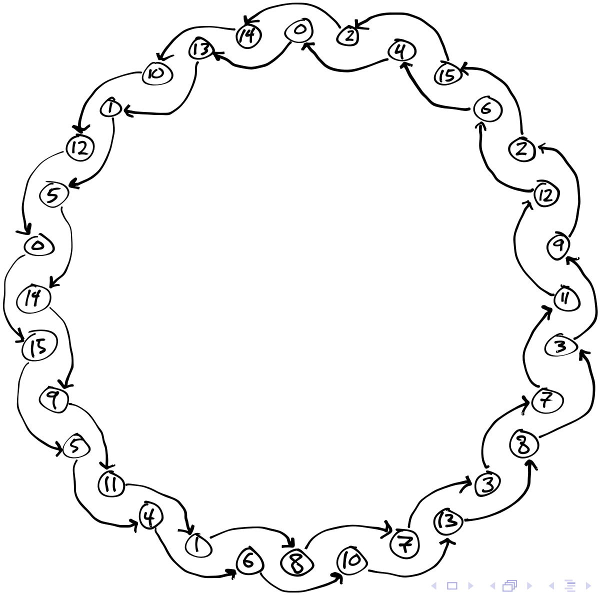
java.util.Random



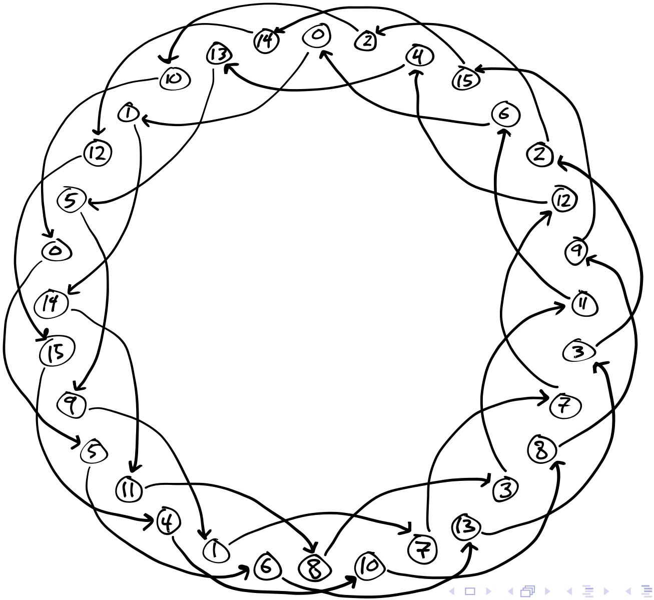
`java.util.Random` as 1 32-count sequence



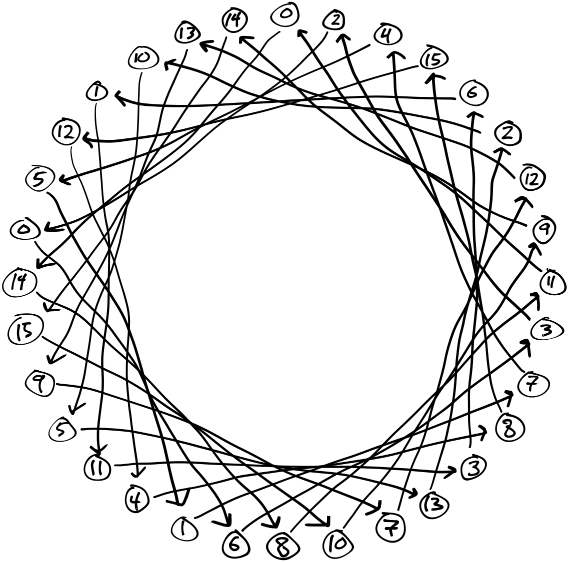
`java.util.Random` as 2 16-count sequences



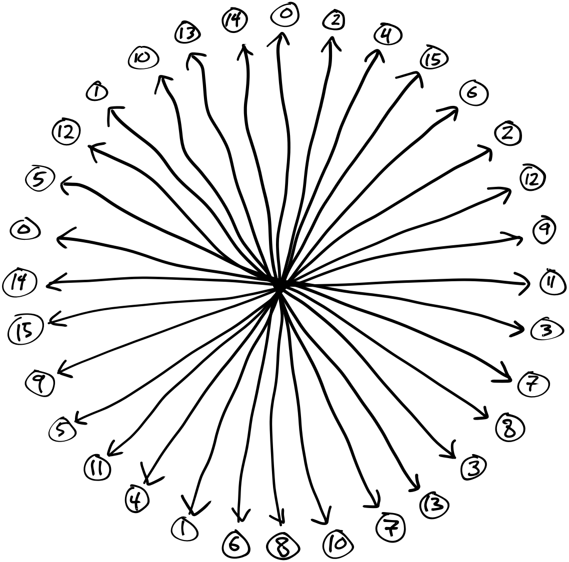
`java.util.Random` as 4 8-count sequences



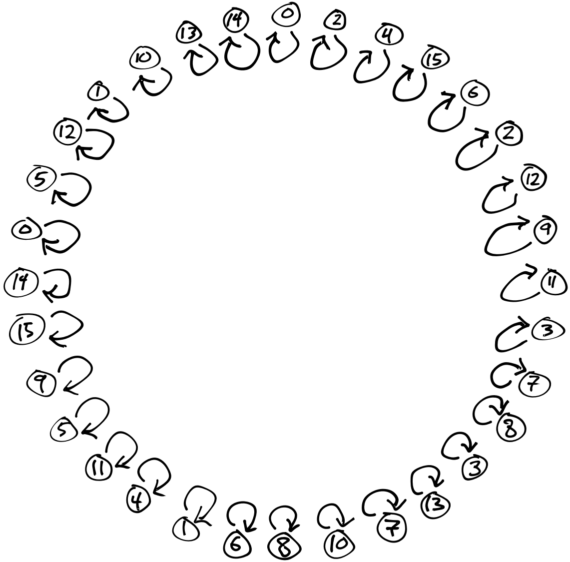
`java.util.Random` as 8 4-count sequences



`java.util.Random` as 16 2-count sequences



java.util.Random as 32 1-count sequences



Haskell's System.Random

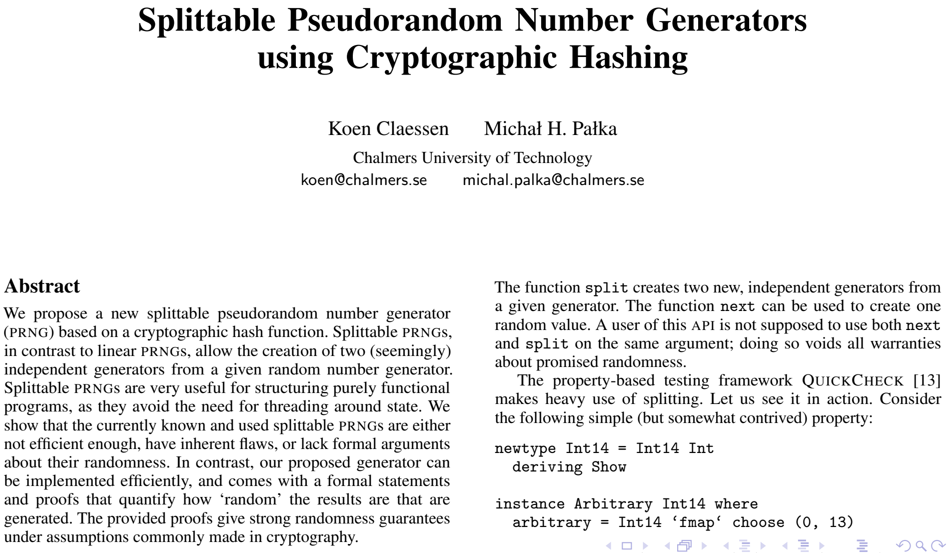
```
stdSplit            :: StdGen -> (StdGen, StdGen)
stdSplit std@(StdGen s1 s2)
                     = (left, right)
                       where
                        -- no statistical foundation for this!
                        left    = StdGen new_s1 t2
                        right   = StdGen t1 new_s2

                        new_s1 | s1 == 2147483562 = 1
                               | otherwise        = s1 + 1

                        new_s2 | s2 == 1          = 2147483398
                               | otherwise        = s2 - 1

                        StdGen t1 t2 = snd (next std)
```

The Lesson

Splittabilizing a linear algorithm can be tricky.

Implementations

High Quality Implementations



Splittable Pseudorandom Number Generators using Cryptographic Hashing

Koen Claessen   Michał H. Pałka
Chalmers University of Technology
koen@chalmers.se   michal.palka@chalmers.se

Abstract

We propose a new splittable pseudorandom number generator (PRNG) based on a cryptographic hash function. Splittable PRNGs, in contrast to linear PRNGs, allow the creation of two (seemingly) independent generators from a given random number generator. Splittable PRNGs are very useful for structuring purely functional programs, as they avoid the need for threading around state. We show that the currently known and used splittable PRNGs are either not efficient enough, have inherent flaws, or lack formal arguments about their randomness. In contrast, our proposed generator can be implemented efficiently, and comes with a formal statements and proofs that quantify how 'random' the results are that are generated. The provided proofs give strong randomness guarantees under assumptions commonly made in cryptography.

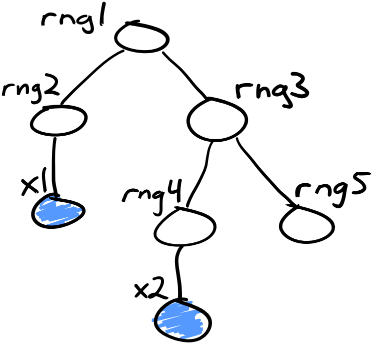
The function `split` creates two new, independent generators from a given generator. The function `next` can be used to create one random value. A user of this API is not supposed to use both `next` and `split` on the same argument; doing so voids all warranties about promised randomness.

The property-based testing framework QUICKCHECK [13] makes heavy use of splitting. Let us see it in action. Consider the following simple (but somewhat contrived) property:

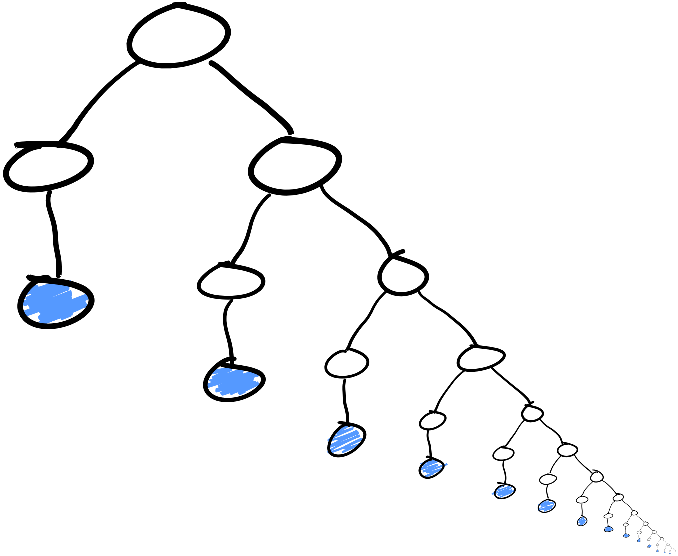
```
newtype Int14 = Int14 Int
  deriving Show

instance Arbitrary Int14 where
  arbitrary = Int14 `fmap` choose (0, 13)

prop shouldFail (   Int14 a) (Int14 b) = a /= b
```

Splitting Tree

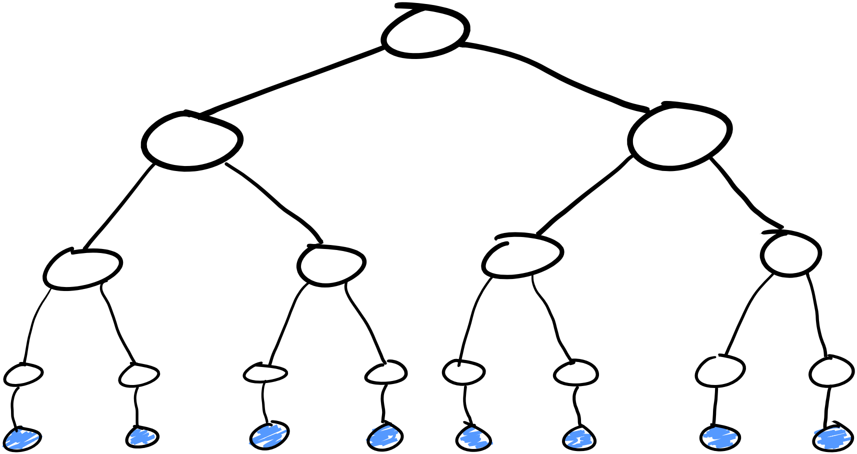
```
(let [rng1 (make-rng seed)
      [rng2 rng3] (split rng1)
      x1 (rand-long rng2)
      [rng4 rng5] (split rng3)
      x2 (rang-long rng4)]
  "hooray")
```



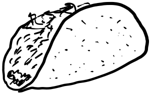
Linear Tree

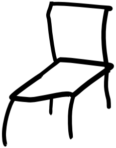


Balanced Tree

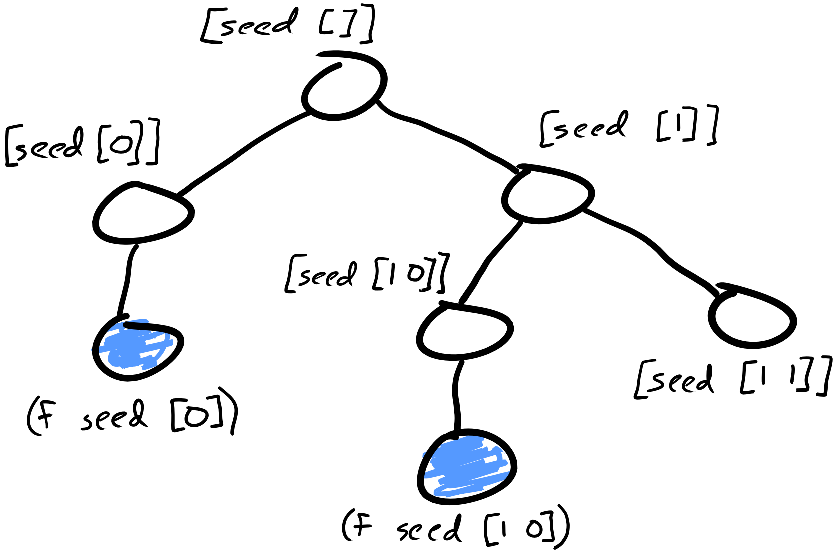


Pseudorandom Function

(f 42) ⇒ 

(f 43) ⇒ 

Tree Path



SHA1Random

```clojure
(deftype SHA1Random [seed path]

  IRandom

  (rand-long [_]
    (bytes->long (sha1 (str seed path))))

  (split [_]
    [(SHA1Random. seed (conj path 0))
     (SHA1Random. seed (conj path 1))]))

(defn sha1-random
  [seed]
  (SHA1Random. seed []))
```

Implementations

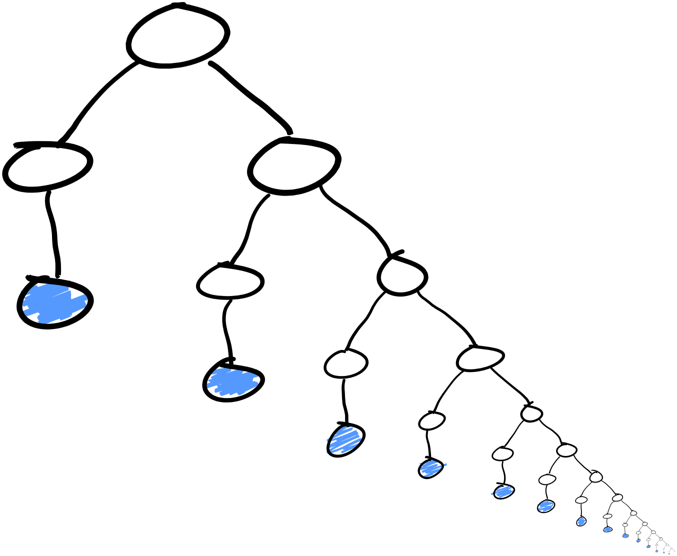
Testing Quality

Dieharder

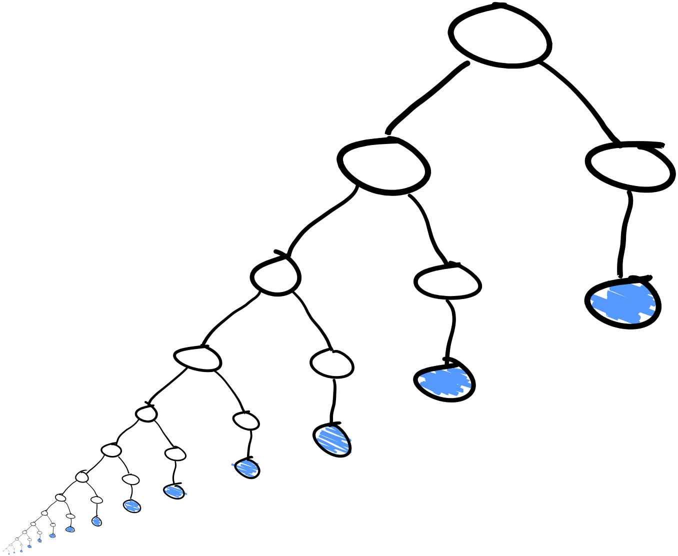
```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown           #
#=============================================================================#

Usage:

dieharder [-a] [-d dieharder test number] [-f filename] [-B]
          [-D output flag [-D output flag] ... ] [-F] [-c separator]
          [-g generator number or -1] [-h] [-k ks_flag] [-l]
          [-L overlap] [-m multiply_p] [-n ntuple]
          [-p number of p samples] [-P Xoff]
          [-o filename] [-s seed strategy] [-S random number seed]
          [-n ntuple] [-p number of p samples] [-o filename]
          [-s seed strategy] [-S random number seed]
          [-t number of test samples] [-v verbose flag]
          [-W weak] [-X fail] [-Y Xtrategy]
          [-x xvalue] [-y yvalue] [-z zvalue]
```

Linearization

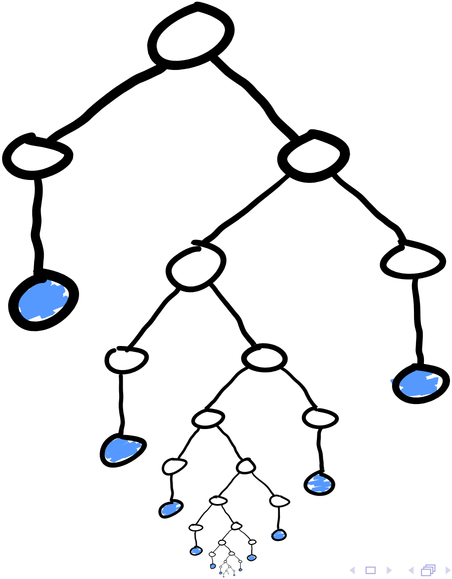
Linearization - Right Linear



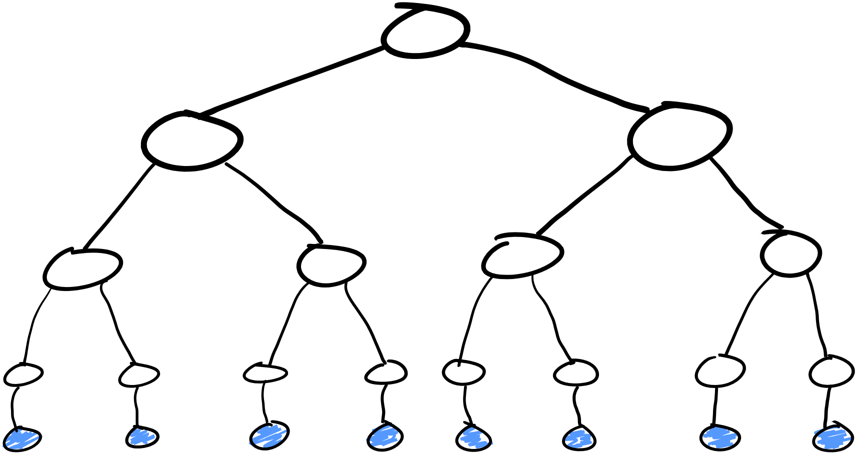
Linearization - Left Linear



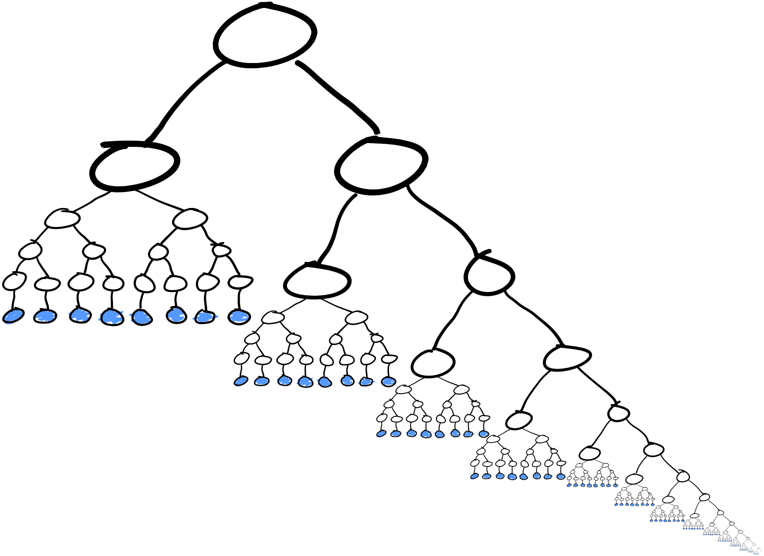
Linearization - Alternating



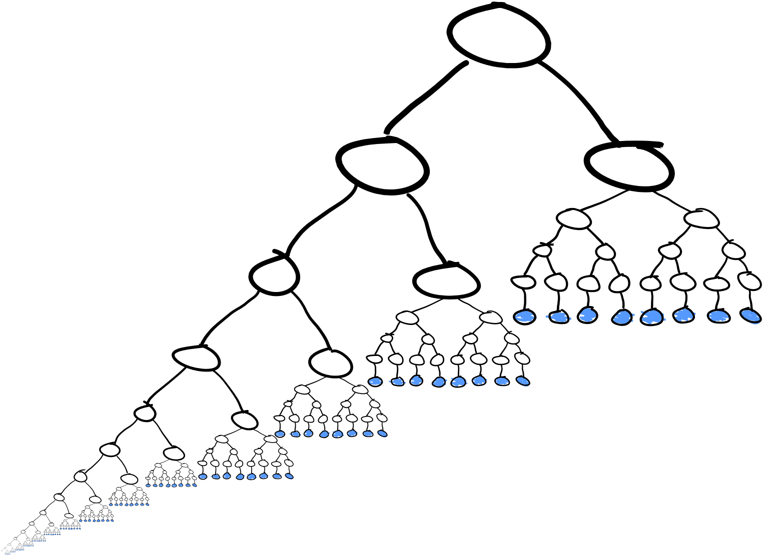
Linearization - Balanced



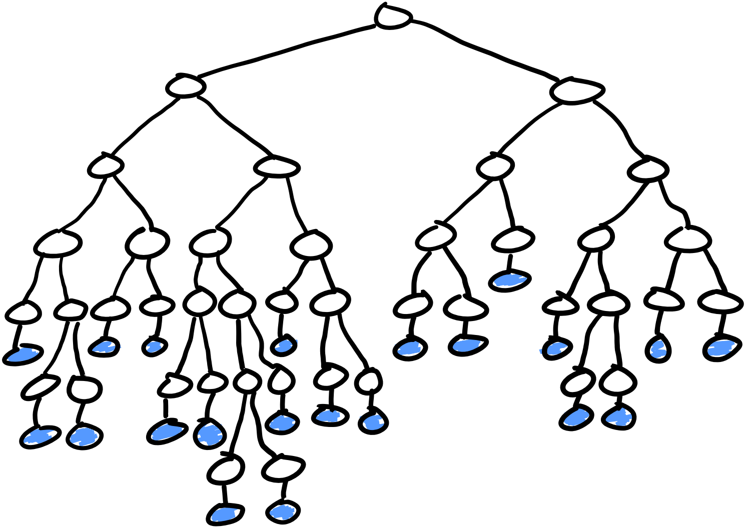
Linearization - Right Lumpy



Linearization - Left Lumpy



Linearization - Fibonacci



Dieharder Results

Algorithm	Linearization	PASSED	WEAK	FAIL
`j.u.Random`	(inherent)	95	13	6
SHA1	left-linear	111	3	0
SHA1	right-linear	112	2	0
SHA1	alternating	114	0	0
SHA1	left-lumpy	110	4	0
SHA1	right-lumpy	112	2	0
SHA1	balanced	112	2	0
SHA1	fibonacci	109	5	0

Implementations

Less Slow Implementations

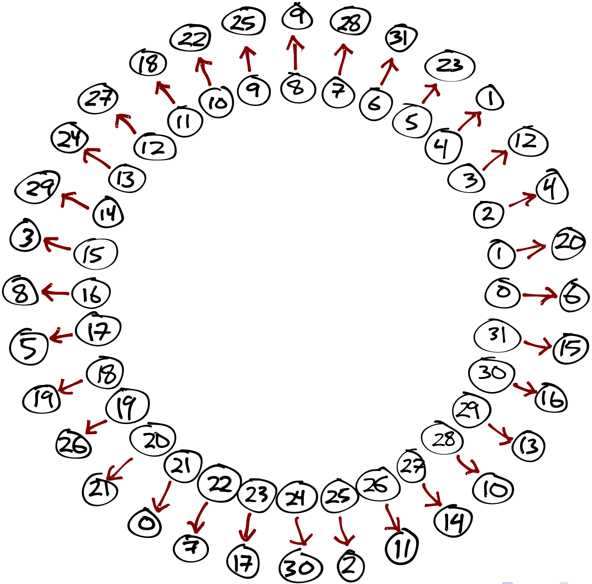
Varying the hash function

Try a faster (noncryptographic?) pseudorandom function, test its quality.

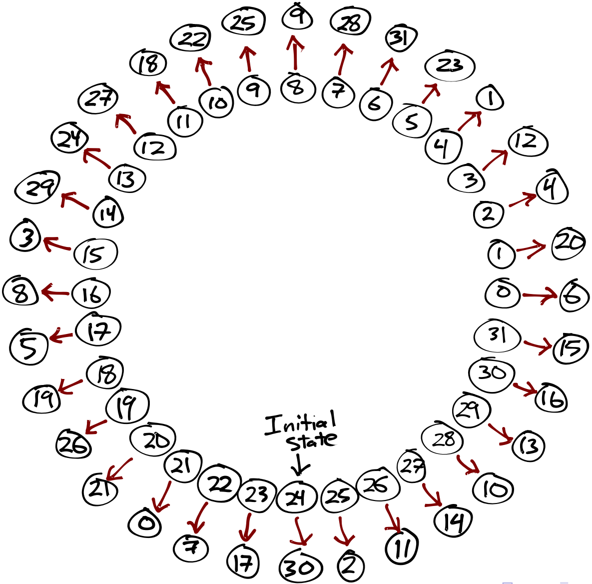
java.util.SplittableRandom

```
public class SplittableRandom{

  public SplittableRandom(long seed){...}

  public long nextLong(){...};

  public SplittableRandom split(){...};

}
```

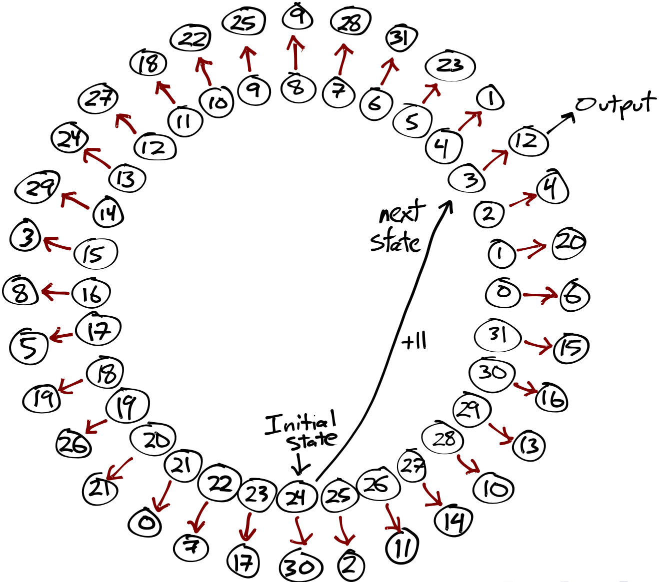
The `java.util.SplittableRandom` Algorithm



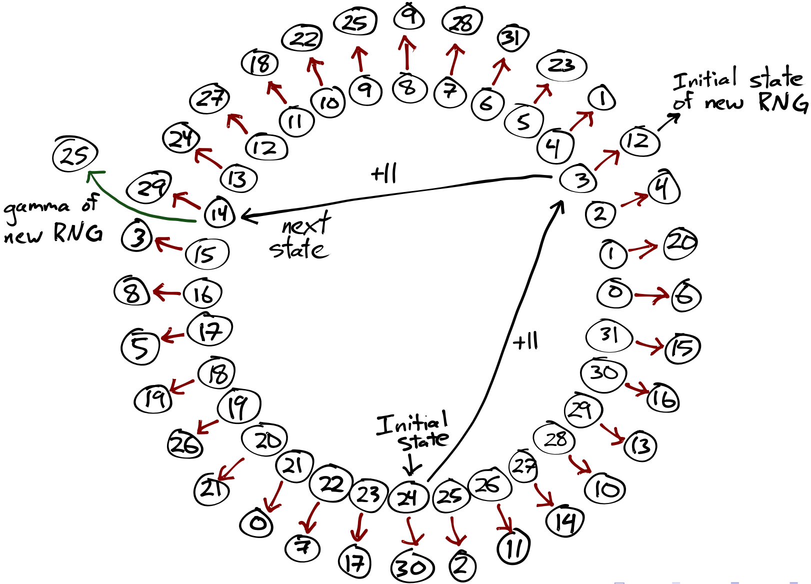
(SplittableRandom. 24)



`(-> 24 (SplittableRandom.) (.nextLong))`



(-> 24 (SplittableRandom.) (.split))

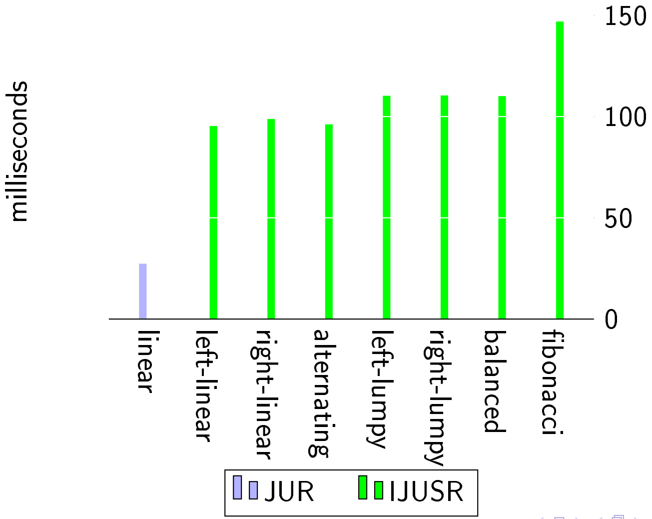


(deftype IJUSR ...)

```clojure
(deftype IJUSR [^long gamma ^long state]
  IRandom
  (rand-long [_]
    (-> state (+ gamma) (mix-64)))
  (split [this]
    (let [state1 (+ gamma state)
          state2 (+ gamma state1)
          new-state (mix-64 state1)
          new-gamma (mix-gamma state2)]
      [(IJUSR. gamma state2)
       (IJUSR. new-gamma new-state)])))
```

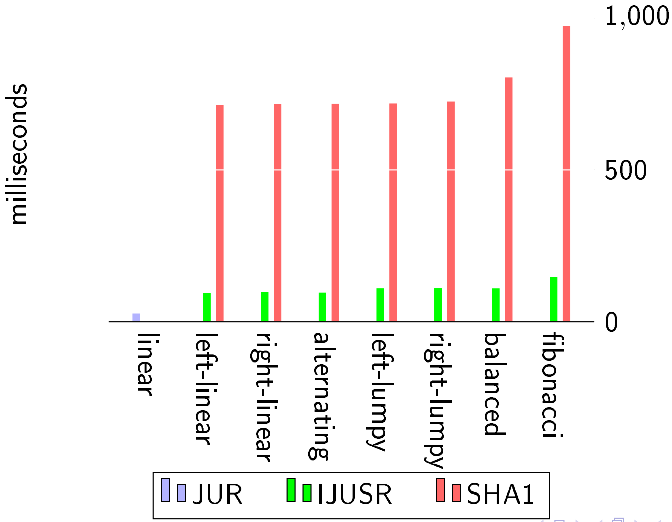
Benchmarks

Criterium tests XORing 1,000,000 random numbers



Benchmarks w/ SHA1

Criterium tests XORing 1,000,000 random numbers



Dieharder Results

Algorithm	Linearization	PASSED	WEAK	FAIL
`j.u.Random`	(inherent)	95	13	6
SHA1	left-linear	111	3	0
SHA1	right-linear	112	2	0
SHA1	alternating	114	0	0
SHA1	left-lumpy	110	4	0
SHA1	right-lumpy	112	2	0
SHA1	balanced	112	2	0
SHA1	fibonacci	109	5	0
IJUSR	left-linear	108	6	0
IJUSR	right-linear	111	3	0
IJUSR	alternating	109	5	0
IJUSR	left-lumpy	113	1	0
IJUSR	right-lumpy	114	0	0
IJUSR	balanced	114	0	0
IJUSR	fibonacci	111	3	0

Implementations

Summary

Okay So

- Linear RNGs cannot be trivially splittabilized
- Recent research provides promising options

Epilogue

Convert test.check to `JavaUtilSplittableRandom`

```
[org.clojure/test.check "0.8.0-ALPHA"]
```

Slowdown

Measuring the slowdown on test.check's own test suite.

`(bench (clojure.test/run-all-tests))`

Before	3.06 ± 0.045 seconds
After	3.56 ± 0.058 seconds

16.3% slower

`lein benchmark-task 20 test`

Before	7.62 ± 0.182 seconds
After	8.34 ± 0.210 seconds

9.3% slower

Empossibleized Future Features

- Parallelizing tests
- Resuming shrinks
- Parallelized shrinks
- Custom shrinking algorithms
- Generating lazy seqs
- Replaying a particular test with a specific "seed"

We Have Come Now To The End

- Splittable RNGs are necessary for composing functional programs
- There are existing splittable algorithms, including `java.util.SplittableRandom`
- Using the `SplittableRandom` algorithm made test.check more robust

Thank You

And also thanks to

- Reid Draper
- Alex Miller

Bibliography

- Claessen, K. ; Palka, M. (2013) "Splittable Pseudorandom Number Generators using Cryptographic Hashing". Proceedings of Haskell Symposium 2013 pp. 47-58.
- Guy L. Steele, Jr., Doug Lea, and Christine H. Flood. 2014. Fast splittable pseudorandom number generators. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14). ACM, New York, NY, USA, 453-472. DOI=10.1145/2660193.2660195 `http://doi.acm.org/10.1145/2660193.2660195`