Class 10: Property-Based Testing

March 28

motivation

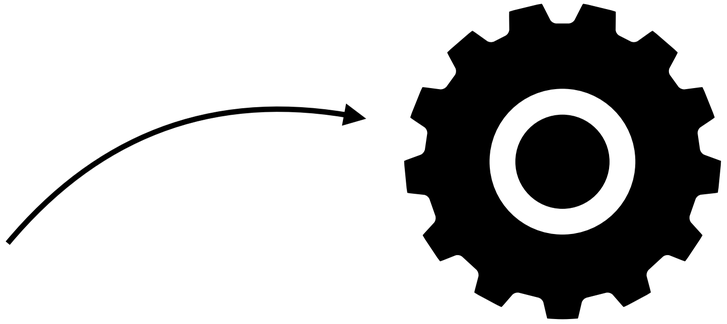
```
sort :: [Int] -> [Int]
```

inputs	**outputs**
[2, 1, 3]	[1, 2, 3]
[]	[]
[3, 3, 1, 1]	[1, 1, 3, 3]
⋮	
[-16, 13, 20, 11, 0, 11, -8, -14, -16, 20, 3, 12, -3, 18, 19, 14]	

```haskell
prop_sort :: [Int] -> Bool
prop_sort xs = _______ (sort xs)
```

```haskell
prop_sort :: [Int] -> Bool
prop_sort xs = ordered (sort xs)


ordered :: [Int] -> Bool
ordered [] = True
ordered [x] = True
ordered (x1 : x2 : xs) = x1 <= x2 && ordered (x2 : xs)
```



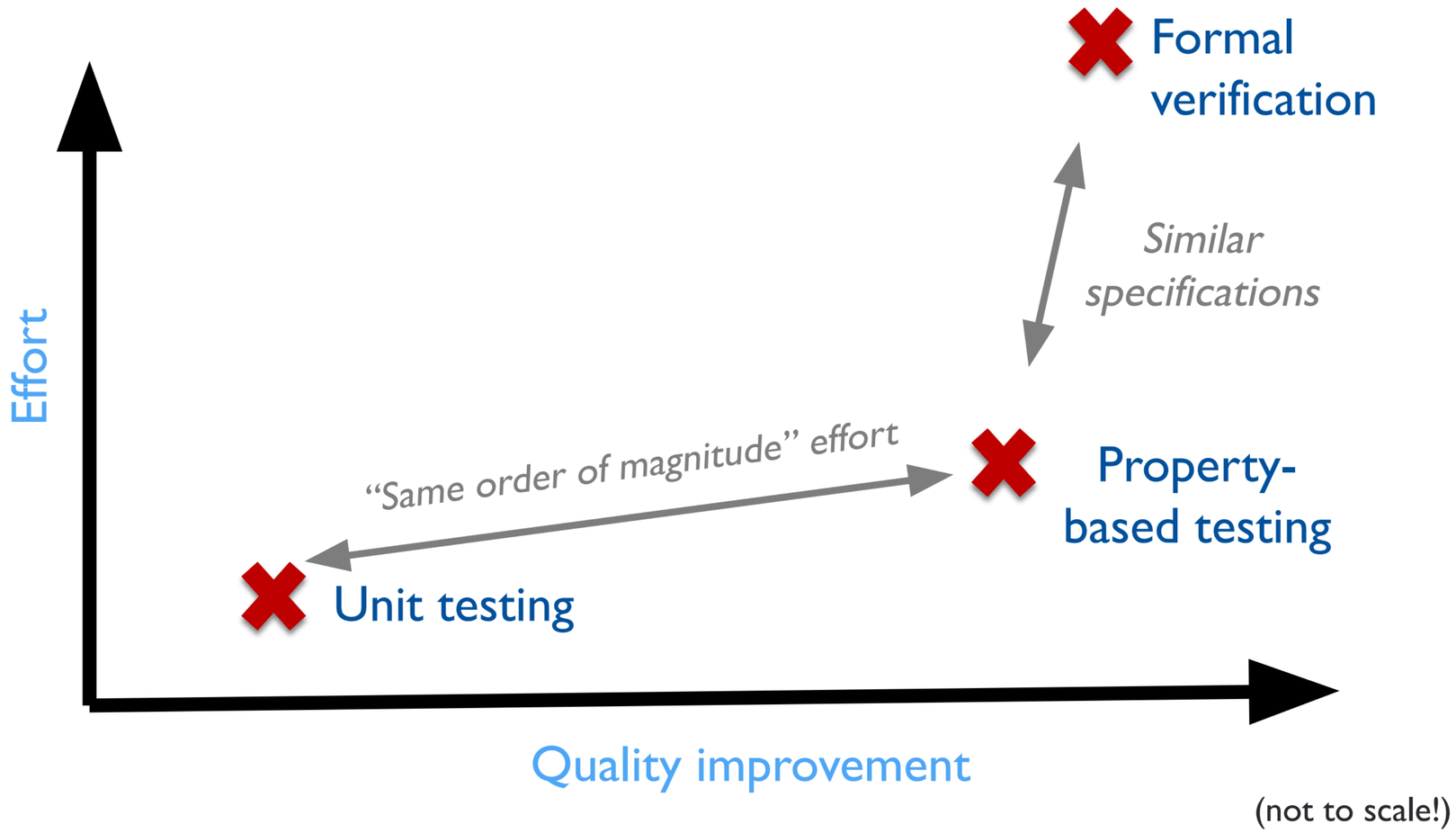
QuickCheck

```haskell
prop_sort :: [Int] -> Bool
prop_sort xs = ordered (sort xs)
```

```haskell
ordered :: [Int] -> Bool
ordered [] = True
ordered [x] = True
ordered (x1 : x2 : xs) = x1 <= x2 && ordered (x2 : xs)
```

(sort exercise)

context



source: Benjamin Pierce's slides

QuickCheck Family

C (theft)	C++ (CppQuickCheck)	Clojure (test.check)	Coq (QuickChick)	F# (FsCheck)
Go (gopter)	Haskell (QuickCheck or Hedgehog)	Java (QuickTheories)	JavaScript (jsverify)	PHP (Eris)
Python (Hypothesis)	Ruby (Rantly)	Rust (Quickcheck)	Scala (ScalaCheck)	Swift (Swiftcheck)

And more!

source: Benjamin Pierce's slides

generating random data

random seed

```
data Gen a = Gen (Rand -> a)
```

```
instance Monad Gen where
    return :: a -> Gen a

    (>>=) :: Gen a -> (a -> Gen b) -> Gen b
```

```haskell
instance Monad Gen where

    return :: a -> Gen a
    return a = Gen (\_ -> a)

    (>>=) :: Gen a -> (a -> Gen b) -> Gen b
```

```haskell
instance Monad Gen where

    return :: a -> Gen a
    return a = Gen (\_ -> a)

    (>>=) :: Gen a -> (a -> Gen b) -> Gen b
    Gen fa >>= k = Gen (\r ->


                    )
```

```haskell
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r

                 )
```

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r
                      (fa r1)
                )
```

```haskell
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r
                    k (fa r1)
                 )
```

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r
         Gen fb = k (fa r1)
               )
```

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r
         Gen fb = k (fa r1)
       in fb r2)
```

```haskell
genBool :: Gen Bool
genBool = return True
```

```haskell
oneof :: [Gen a] -> Gen a

genBool :: Gen Bool
genBool = oneof [return True, return False]
```

```
genTwoBool :: Gen (Bool, Bool)
genTwoBool = do
  b1 <- genBool
  b2 <- genBool
  return (b1, b2)
```

```
class Aribtrary a where
   arbitrary :: Gen a
```

```haskell
instance Aribtrary Bool where
   arbitrary :: Gen Bool
   arbitrary = oneof [return True, return False]
```

```haskell
instance (Arbitrary a, Arbitrary b) => Aribtrary (a, b) where
   arbitrary :: Gen (a, b)
   arbitrary = do
     a <- (arbitrary :: Gen a)
     b <- (arbitrary :: Gen b)
     return (a, b)
```

```haskell
instance Arbitrary a => Aribtrary [a] where
   arbitrary :: Gen [a]
   arbitrary =
     oneof
       [ return [],
         do
           x <- arbitrary
           xs <- arbitrary
           return (x : xs)
       ]
```

```haskell
instance Arbitrary a => Aribtrary [a] where
   arbitrary :: Gen [a]
   arbitrary =
     frequency
       [ (1, return []),
         (4, do
           x <- arbitrary
           xs <- arbitrary
           return (x : xs)
         )
       ]
```

(expr exercise)

size

```
data Gen a = Gen (Rand -> Int -> a)
```

```haskell
genNum :: Gen Expr
genNum = do
  n <- arbitrary
  return (Num n)
```

```haskell
genExpr :: Int -> Gen Expr
```

```haskell
genNum :: Gen Expr
genNum = do
  n <- arbitrary
  return (Num n)
```

```haskell
genExpr :: Int -> Gen Expr
genExpr 0 = genNum
```

```haskell
genNum :: Gen Expr
genNum = do
  n <- arbitrary
  return (Num n)
```

```haskell
genExpr :: Int -> Gen Expr
genExpr 0 = genNum
genExpr n =
  frequency
    [(1, genNum),
     (4, do
       e1 <-
       e2 <-
       return (Add e1 e2)
     )]

genNum :: Gen Expr
genNum = do
  n <- arbitrary
  return (Num n)
```

```haskell
genExpr :: Int -> Gen Expr
genExpr 0 = genNum
genExpr n =
  frequency
    [(1, genNum),
     (4, do
       e1 <- genExpr (n `div` 2)
       e2 <- genExpr (n `div` 2)
       return (Add e1 e2)
     )]

genNum :: Gen Expr
genNum = do
  n <- arbitrary
  return (Num n)
```

writing properties

```
quickCheck :: Testable prop => prop -> IO ()
```

```
instance Testable Bool

instance (Arbitrary a, Show a, Testable prop)
  => Testable (a -> prop)
```

(remove example)

PBT at Penn

Generating Good Generators for Inductive Relations

LEONIDAS LAMPROPOULOS, University of Pennsylvania, USA
ZOE PARASKEVOPOULOU, Princeton University, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

Parsing Randomness

HARRISON GOLDSTEIN, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

Some Problems with Properties

A Study on Property-Based Testing in Industry

HARRISON GOLDSTEIN, University of Pennsylvania, USA
JOSEPH W. CUTLER, University of Pennsylvania, USA
ADAM STEIN, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
ANDREW HEAD, University of Pennsylvania, USA