Edit distance

smallest number of inserts/deletes to turn arg#1 into arg#2

```
dist :: Eq a => [a] -> [a] -> Int

Main> dist "abcd" "xaby"
4

Main> dist "" "monkey"
6

Main> dist "Haskell" ""
7

Main> dist "hello" "hello"
0
```

Edit distance implementation

```
dist :: Eq a => [a] -> [a] ->
dist []     ys     = lengt
dist xs     []     = length
dist (x:xs) (y:ys)
   | x == y          = dist xs ys
   | otherwise       = (1 + dist (     ) ys)
               `min` (1 + dist xs (y:ys))
```

challenge #0:
implement a polynomial
time version

two recursive calls:
exponential time

either insert y or
delete x

How to test? -- ”Test Oracle”

- Formal specification
- Executable
- Efficient (polynomial time)

think QuickCheck

comparing against naive dist is no good...

challenge #1: find an practical way to test your implementation!

(answer)

An efficient dist

dynamic programming

```
dist :: Eq a => [a] -> [a] -> Int
dist xs ys = head (dists xs ys)

dists :: Eq a => [a] -> [a] -> [Int]
dists []     ys = [n,n-1..0] where n = length ys
dists (x:xs) ys = line x ys (dists xs ys)

line :: Eq a => a -> [a] -> [Int] -> [Int]
line x []     [d]     = [d+1]
line x (y:ys) (d:ds)
  | x == y            = head ds : ds'
  | otherwise         = (1+(d`min`
 where
  ds' = line x ys ds
```

testing
upper-bound: easy,
lower-bound: hard

Naive dist

```
dist :: Eq a => [a] -> [a] -> Int
dist []      ys      = length ys

dist xs      []      = length xs

dist (x:xs) (y:ys)
  | x == y           = dist xs ys

dist (x:xs) (y:ys)
  | otherwise        = (1 + dist (x:xs) ys)
                `min` (1 + dist xs (y:ys))
```

base case #1

base case #2

step case #1

step case #2

”Inductive Testing”

```
prop_BaseXs (ys :: String) =
  dist [] ys == length ys

prop_BaseYs (xs :: String) =
  dist xs [] == length xs

prop_StepSame x xs (ys :: String) =
  dist (x:xs) (x:ys) == dist xs ys

prop_StepDiff x y xs (ys :: String) =
  x /= y ==>
    dist (x:xs) (y:ys) == (1 + dist (x:xs) ys) `min`
                          (1 + dist xs (y:ys))
```

specialization

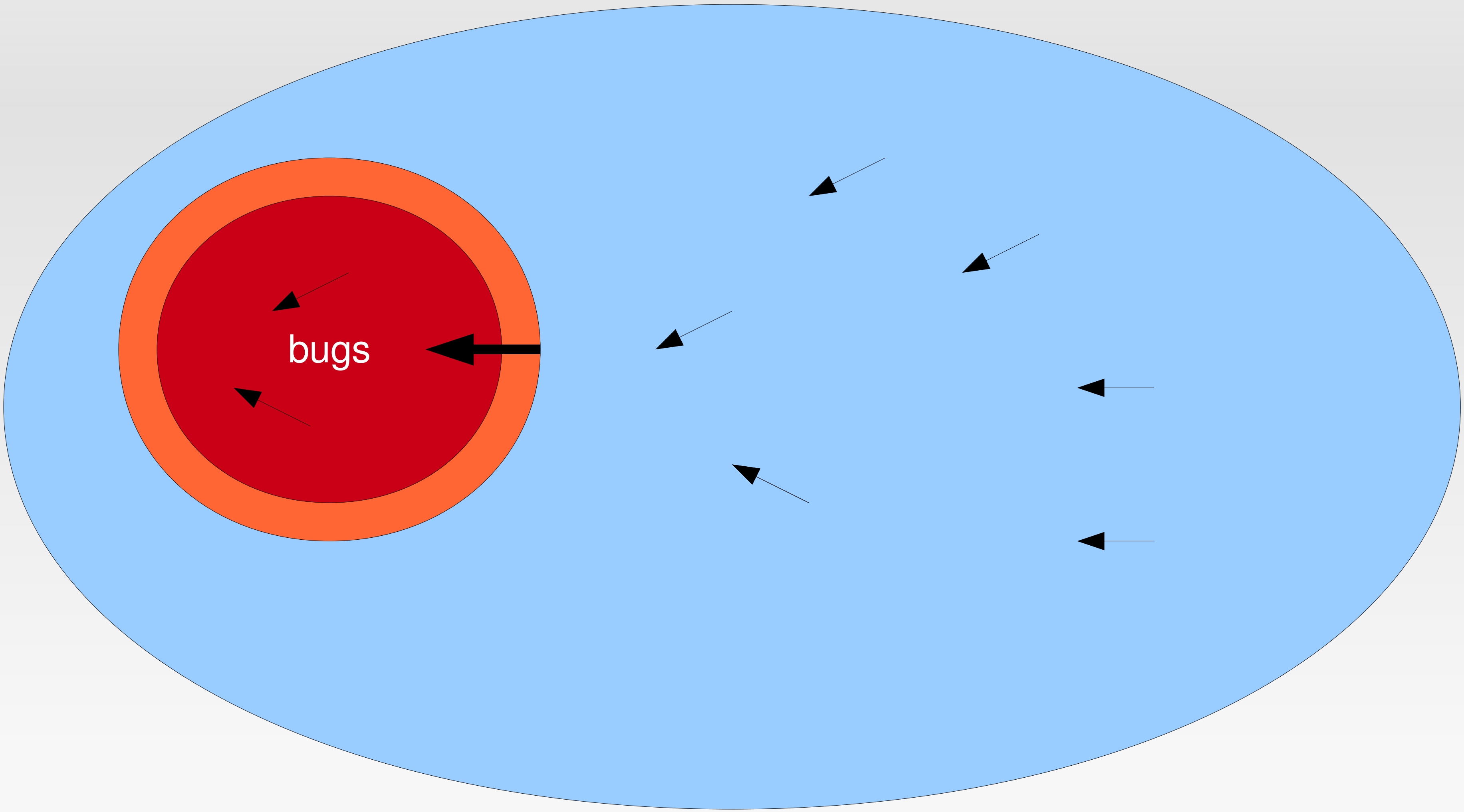
(Alternative)

```
distFix :: Eq a => ([a] -> [a] -> Int)
                -> ([a] -> [a] -> Int)
distFix f []     ys     = length ys
distFix f xs     []     = length xs
distFix f (x:xs) (y:ys)
  | x == y              = f xs ys
  | otherwise           = (1 + f (x:xs) ys)
                     `min` (1 + f xs (y:ys))
```

no recursion

```
prop_Dist xs (ys :: String) =
  dist xs ys == distFix dist xs ys
```

What is happening?



Applications

- Search algorithms
  - SAT-solvers
  - other kinds of solvers
- Optimization algorithms
  - LP-solvers
  - (edit distance)
- Symbolic algorithms?
  - substitution, unification, anti-unification, ...