Informatics 1

Functional Programming Lectures 11 and 12

Monday 3 and Tuesday 4 November 2014

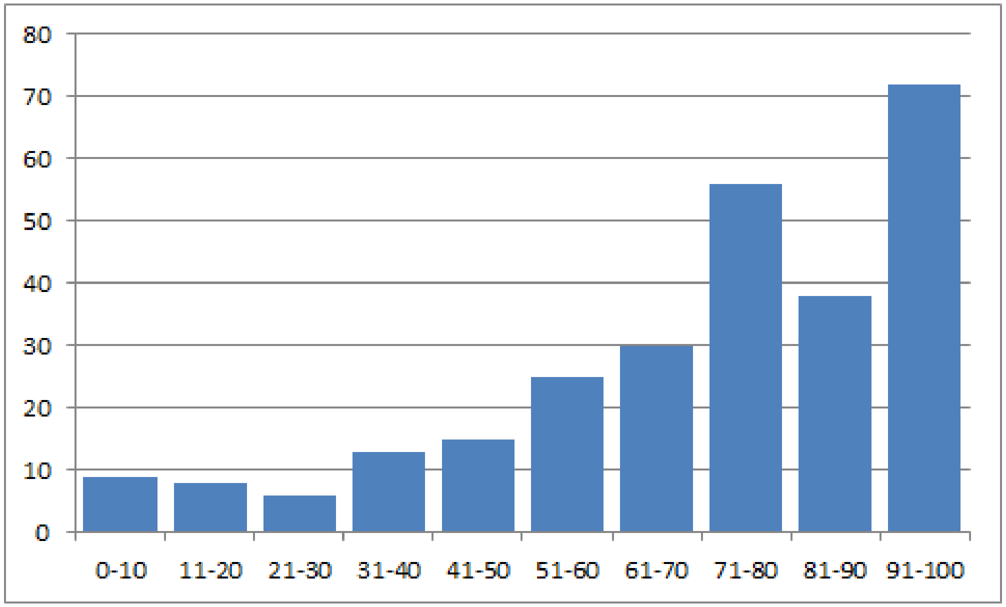
Abstract Types

Don Sannella

University of Edinburgh

Class test

Class test marks



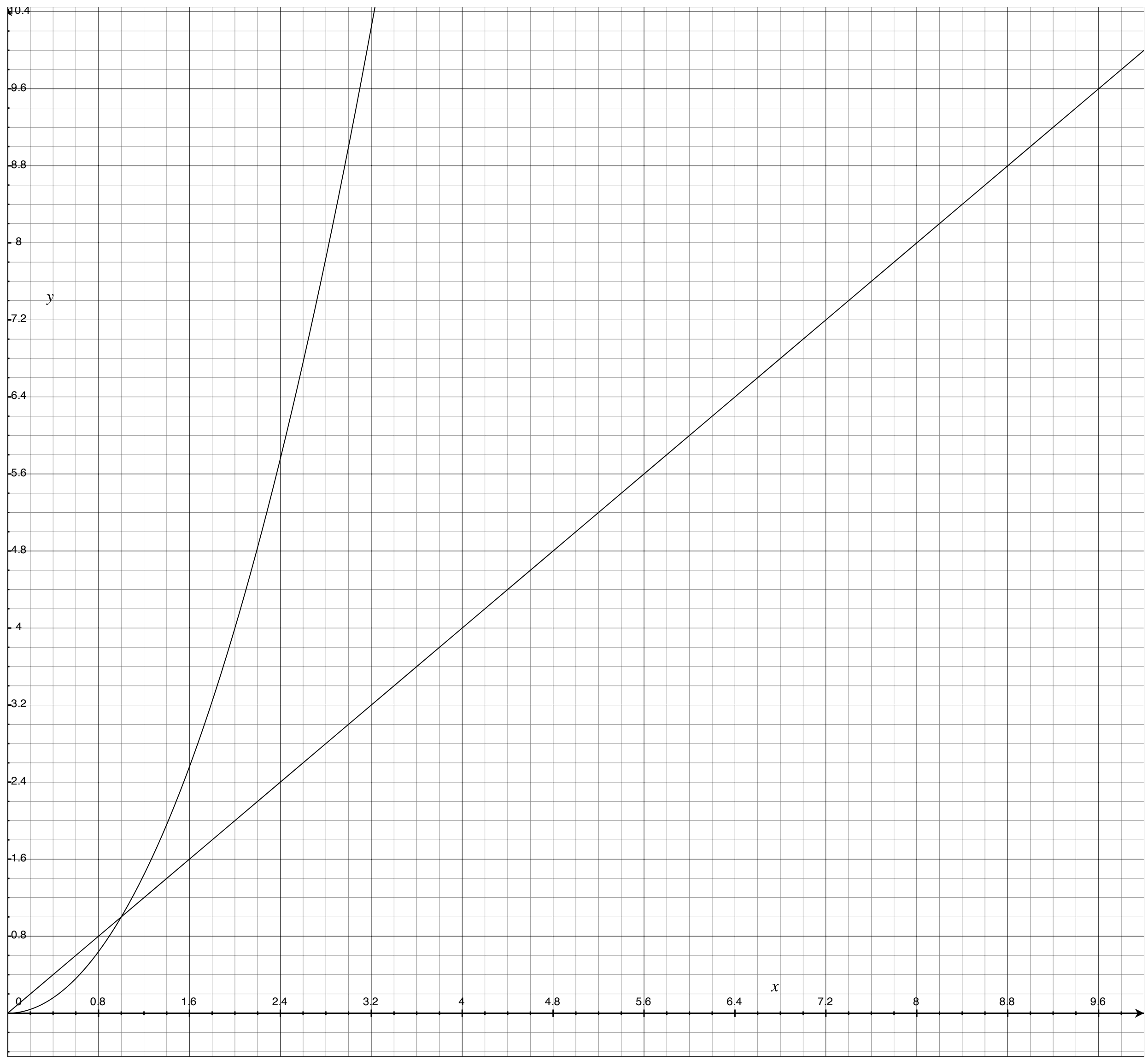
Revision tutorials

- *In addition* to the usual weekly tutorial
- For those who want extra help; no need to sign up
- Every Wednesday 2–3pm in AT 5.05
- Do the extra tutorial exercises on the course webpage before the tutorial, and bring your attempt to the tutorial

Part I

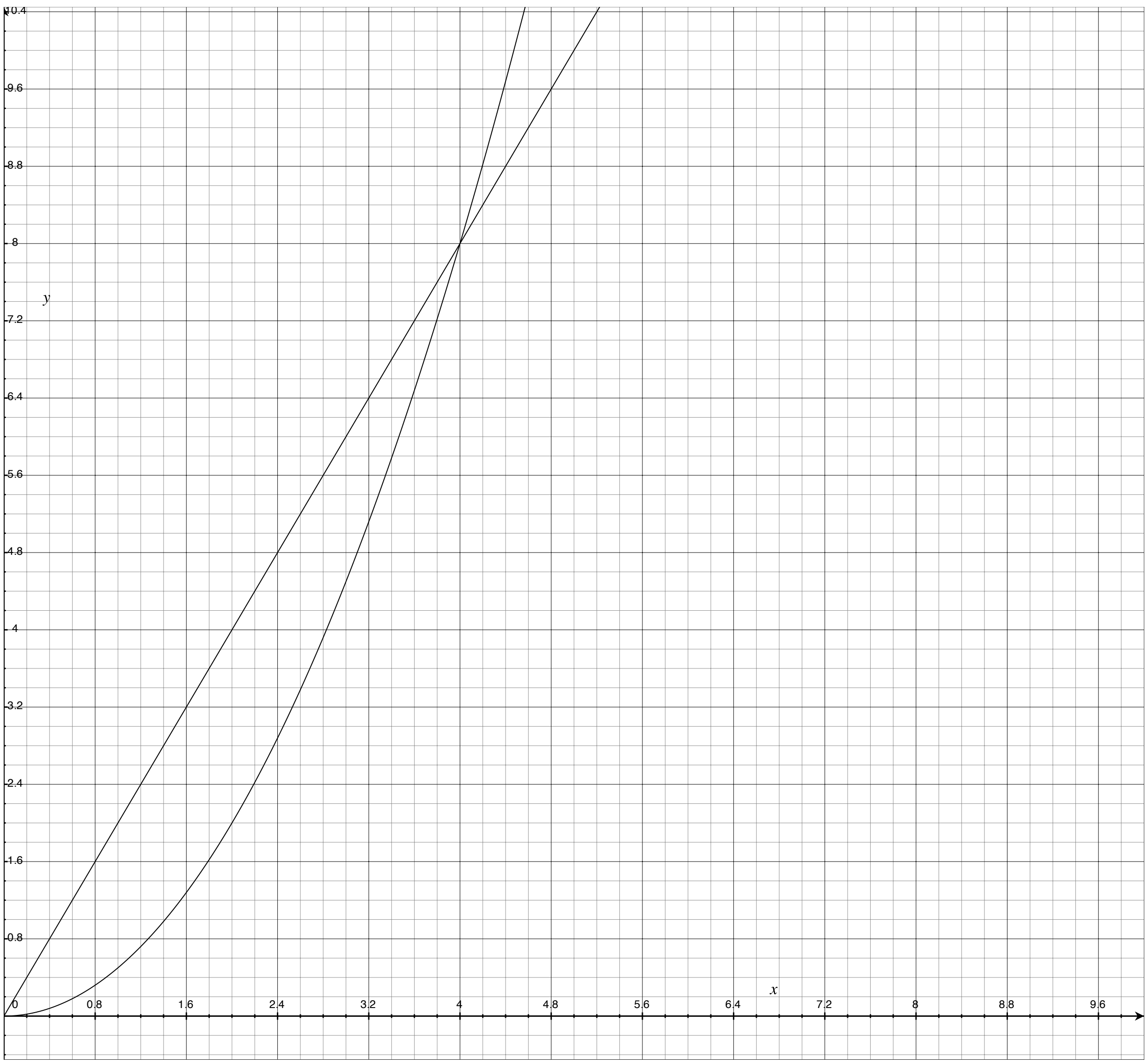
Complexity

$t = n$ vs $t = n^2$



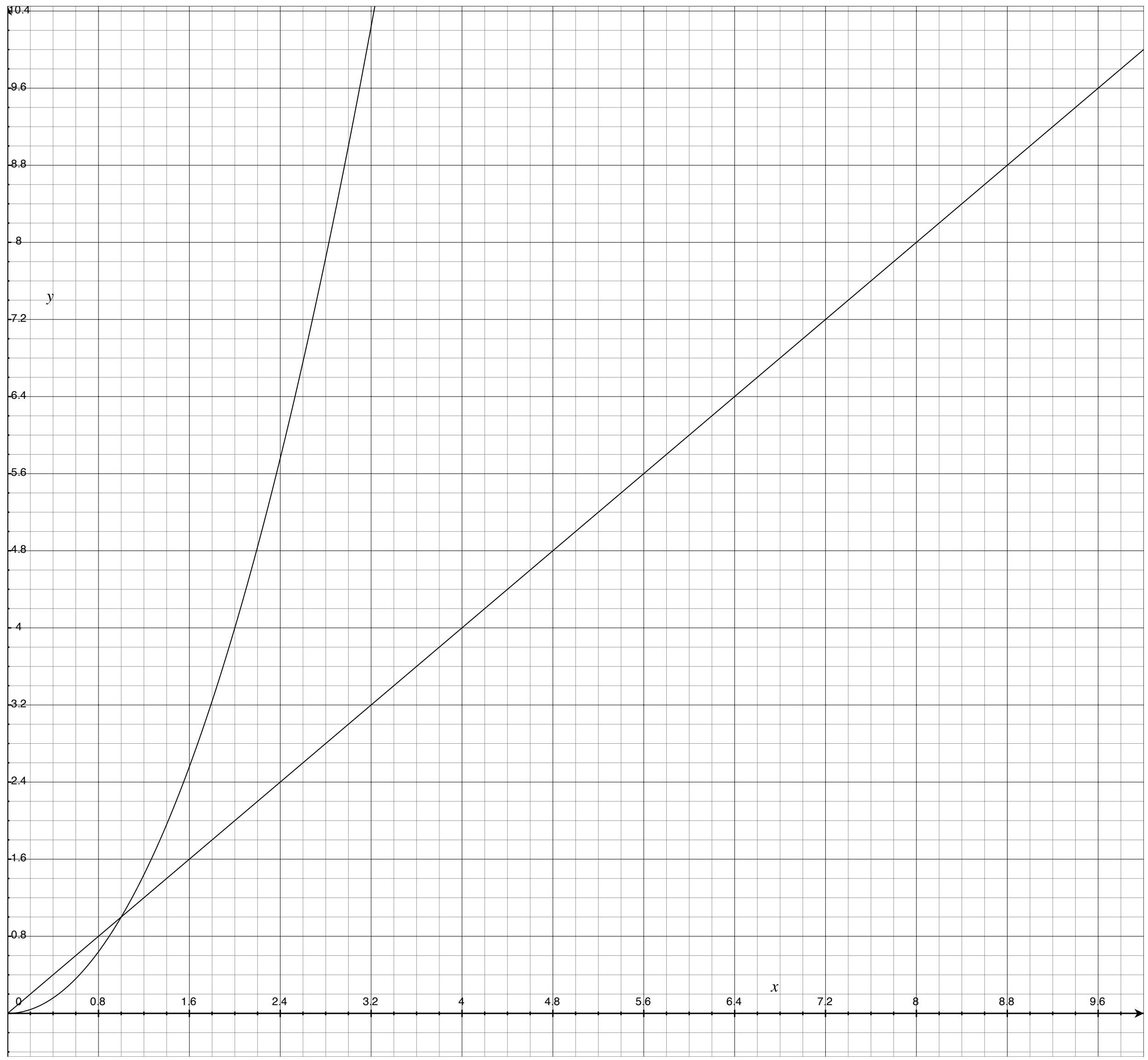
10.4
9.6
8.8
8
7.2
6.4
5.6
4.8
4
3.2
2.4
1.6
0.8
y
0
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
7.2
8
8.8
9.6
x

$t = 2n$ vs $t = 0.5n^2$



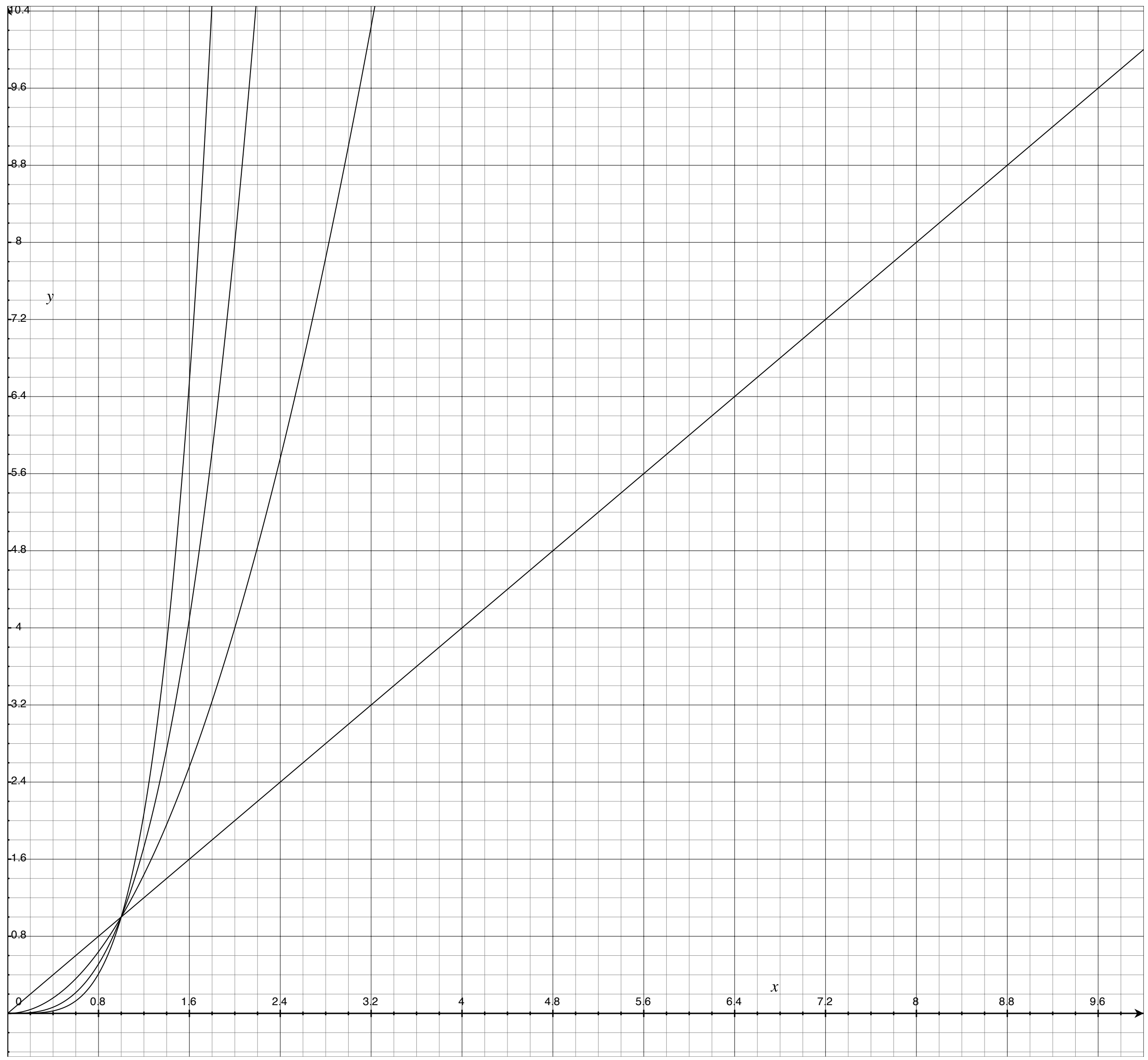
10.4
9.6
8.8
8
y
7.2
6.4
5.6
4.8
4
3.2
2.4
1.6
0.8
x
0
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
7.2
8
8.8
9.6

$O(n)$ vs $O(n^2)$

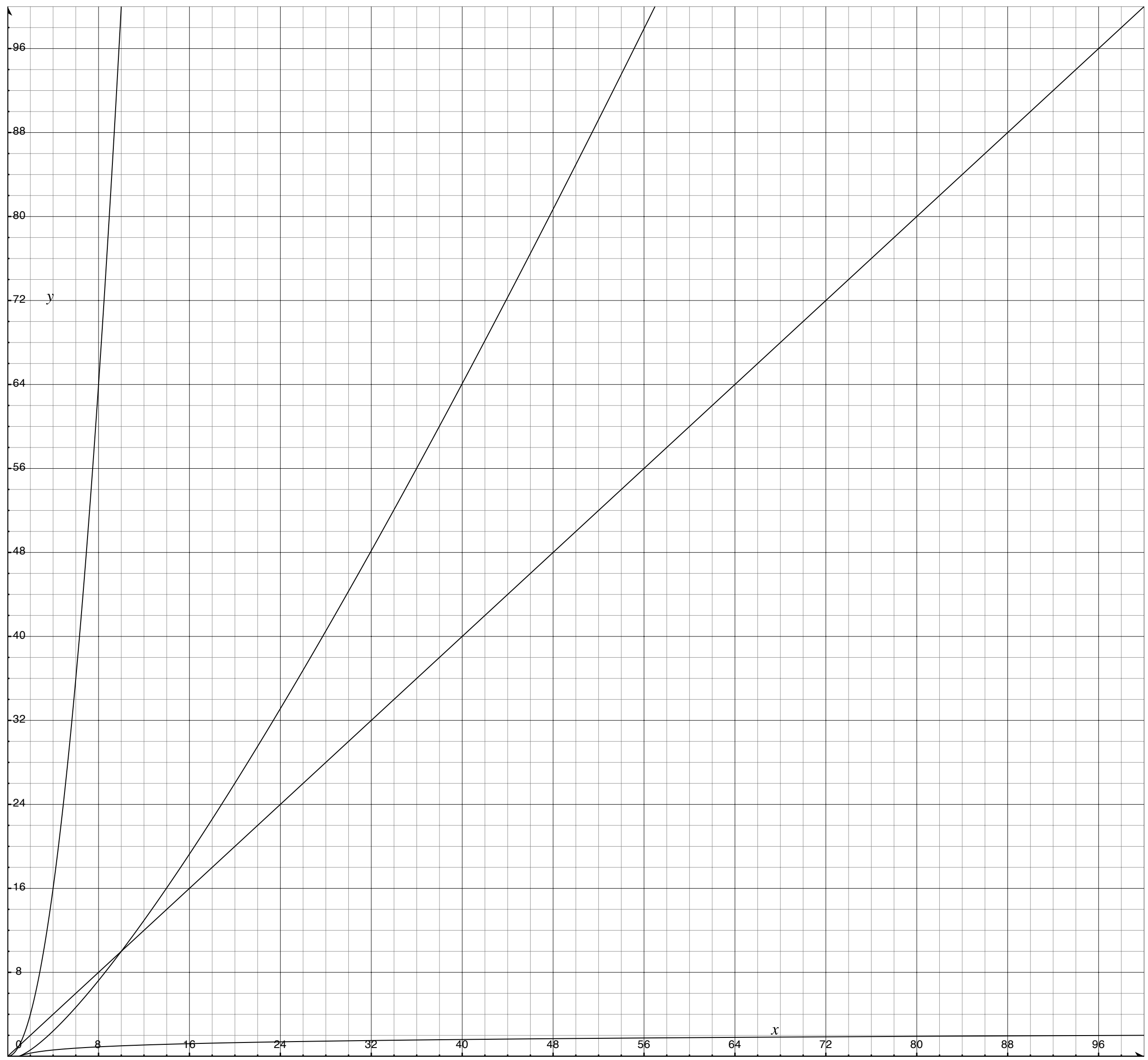


10.4
9.6
8.8
8
7.2
6.4
5.6
4.8
4
3.2
2.4
1.6
0.8
0
y
x
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
7.2
8
8.8
9.6

$O(n), O(n^2), O(n^3), O(n^4)$



$O(\log n), O(n), O(n \log n), O(2^n)$



Part II

Sets as lists without abstraction

ListUnabs.hs (1)

```
module ListUnabs
  (Set,nil,insert,set,element,equal,check) where
import Test.QuickCheck

type Set a = [a]

nil :: Set a
nil =  []

insert :: a -> Set a -> Set a
insert x xs  =  x:xs

set :: [a] -> Set a
set xs  =  xs
```

ListUnabs.hs (2)

```
element :: Eq a => a -> Set a -> Bool
x `element` xs  =  x `elem` xs

equal :: Eq a => Set a -> Set a -> Bool
xs `equal` ys  =  xs `subset` ys && ys `subset` xs
  where
  xs `subset` ys  =  and [ x `elem` ys | x <- xs ]
```

ListUnabs.hs (3)

```
prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_element

-- Prelude ListUnabs> check
-- +++ OK, passed 100 tests.
```

ListUnabsTest.hs

```
module ListUnabsTest where
import ListUnabs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

breakAbstraction :: Set a -> a
breakAbstraction =  head

-- not a function!
-- head (set [1,2,3]) == 1 /= 3 == head (set [3,2,1])
```

Part III

Sets as *ordered* lists without abstraction

OrderedListUnabs.hs (1)

```
module OrderedListUnabs
  (Set,nil,insert,set,element,equal,check) where

import Data.List(nub,sort)
import Test.QuickCheck

type Set a = [a]

invariant :: Ord a => Set a -> Bool
invariant xs  =
  and [ x < y | (x,y) <- zip xs (tail xs) ]
```

OrderedListUnabs.hs (2)

```haskell
nil :: Set a
nil =  []

insert :: Ord a => a -> Set a -> Set a
insert x []                 =  [x]
insert x (y:ys) | x < y     =  x : y : ys
                | x == y    =  y : ys
                | x > y     =  y : insert x ys

set :: Ord a => [a] -> Set a
set xs  =  nub (sort xs)
```

OrderedListUnabs.hs (3)

```
element :: Ord a => a -> Set a -> Bool
x `element` []                    =  False
x `element` (y:ys) | x < y        =  False
                   | x == y       =  True
                   | x > y        =  x `element` ys

equal :: Eq a => Set a -> Set a -> Bool
xs `equal` ys  =  xs == ys
```

OrderedListUnabs.hs (4)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

Prelude OrderedListUnabs> check
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

OrderedListUnabsTest.hs

```
module OrderedListUnabsTest where
import OrderedListUnabs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

breakAbstraction :: Set a -> a
breakAbstraction =  head
-- now it's a function
-- head (set [1,2,3]) == 1 == head (set [3,2,1])

badtest :: Int -> Bool
badtest n =
  s `equal` t
  where
  s = [1,2..n]      -- no call to set!
  t = [n,n-1..1]    -- no call to set!
```

Part IV

Sets as ordered trees without abstraction

TreeUnabs.hs (1)

```
module TreeUnabs
  (Set(Nil,Node),nil,insert,set,element,equal,check) where
import Test.QuickCheck

data  Set a  =  Nil | Node (Set a) a (Set a)

list :: Set a -> [a]
list Nil               =  []
list (Node l x r)  =  list l ++ [x] ++ list r

invariant :: Ord a => Set a -> Bool
invariant Nil    =  True
invariant (Node l x r)  =
  invariant l && invariant r &&
  and [ y < x | y <- list l ] &&
  and [ y > x | y <- list r ]
```

TreeUnabs.hs (2)

```
nil :: Set a
nil  =  Nil

insert :: Ord a => a -> Set a -> Set a
insert x Nil  =  Node Nil x Nil
insert x (Node l y r)
  | x == y     =  Node l y r
  | x < y      =  Node (insert x l) y r
  | x > y      =  Node l y (insert x r)

set :: Ord a => [a] -> Set a
set  =  foldr insert nil
```

TreeUnabs.hs (3)

```
element :: Ord a => a -> Set a -> Bool
x `element` Nil   =  False
x `element` (Node l y r)
  | x == y     =  True
  | x < y      =  x `element` l
  | x > y      =  x `element` r

equal :: Ord a => Set a -> Set a -> Bool
s `equal` t  =  list s == list t
```

TreeUnabs.hs (4)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

-- Prelude TreeUnabs> check
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```

TreeUnabsTest.hs

```
module TreeUnabsTest where
import TreeUnabs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

badtest :: Bool
badtest =
  s `equal` t
  where
  s = set [1,2,3]
  t = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
  -- breaks the invariant!
```

Part V

Sets as *balanced* trees without abstraction

BalancedTreeUnabs.hs (1)

```
module BalancedTreeUnabs
  (Set(Nil,Node),nil,insert,set,element,equal,check) where
import Test.QuickCheck

type  Depth  =  Int
data  Set a  =  Nil | Node (Set a) a (Set a) Depth

node :: Set a -> a -> Set a -> Set a
node l x r  =  Node l x r (1 + (depth l `max` depth r))

depth :: Set a -> Int
depth Nil  =  0
depth (Node _ _ _ d) = d
```

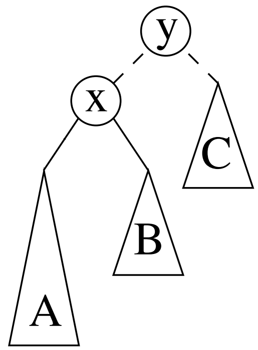
BalancedTreeUnabs.hs (2)

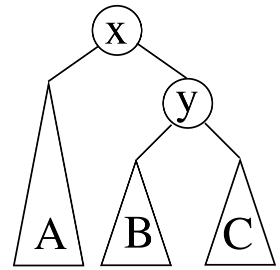
```
list :: Set a -> [a]
list Nil                =  []
list (Node l x r _)     =  list l ++ [x] ++ list r

invariant :: Ord a => Set a -> Bool
invariant Nil    =  True
invariant (Node l x r d)   =
  invariant l && invariant r &&
  and [ y < x | y <- list l ] &&
  and [ y > x | y <- list r ] &&
  abs (depth l - depth r) <= 1 &&
  d == 1 + (depth l `max` depth r)
```

BalancedTreeUnabs.hs (3)

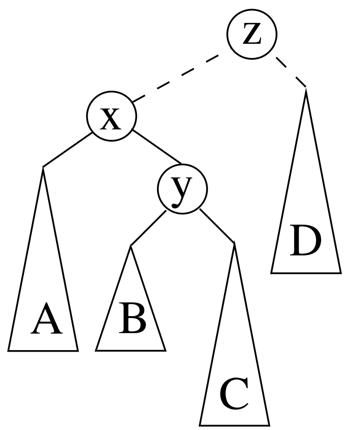
```
nil :: Set a
nil  =  Nil

insert :: Ord a => a -> Set a -> Set a
insert x Nil  =  node nil x nil
insert x (Node l y r _)
  | x == y      =  node l y r
  | x < y       =  rebalance (node (insert x l) y r)
  | x > y       =  rebalance (node l y (insert x r))

set :: Ord a => [a] -> Set a
set  =  foldr insert nil
```

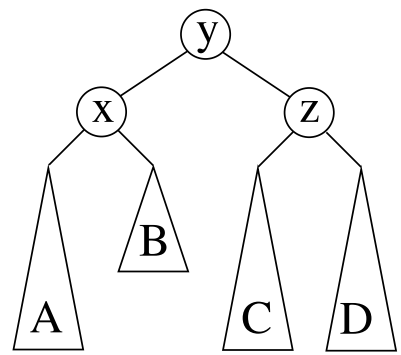
Rebalancing






```
Node (Node a x b) y c   -->   Node a x (Node b y c)
```






```
Node (Node a x (Node b y c) z d)
            --> Node (Node a x b) y (Node c z d)
```

BalancedTreeUnabs.hs (4)

```
rebalance :: Set a -> Set a
rebalance (Node (Node a x b _) y c _)
  | depth a >= depth b && depth a > depth c
  = node a x (node b y c)
rebalance (Node a x (Node b y c _) _)
  | depth c >= depth b && depth c > depth a
  = node (node a x b) y c
rebalance (Node (Node a x (Node b y c _) _) z d _)
  | depth (node b y c) > depth d
  = node (node a x b) y (node c z d)
rebalance (Node a x (Node (Node b y c _) z d _) _)
  | depth (node b y c) > depth a
  = node (node a x b) y (node c z d)
rebalance a  =  a
```

BalancedTreeUnabs.hs (5)

```
element :: Ord a => a -> Set a -> Bool
x `element` Nil  =  False
x `element` (Node l y r _)
  | x == y      =  True
  | x < y       =  x `element` l
  | x > y       =  x `element` r

equal :: Ord a => Set a -> Set a -> Bool
s `equal` t  =  list s == list t
```

BalancedTreeUnabs.hs (6)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

-- Prelude BalancedTreeUnabs> check
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```

BalancedTreeUnabsTest.hs

```haskell
module BalancedTreeUnabsTest where
import BalancedTreeUnabs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

badtest :: Bool
badtest =
  s `equal` t
  where
  s = set [1,2,3]
  t = (Node Nil 1 (Node Nil 2 (Node Nil 3 Nil 1) 2) 3)
  -- breaks the invariant!
```

Part VI

Complexity, revisited

Summary

	insert	set	element	equal
List	$O(1)$	$O(1)$	$O(n)$	$O(n^2)$
OrderedList	$O(n)$	$O(n \log n)$	$O(n)$	$O(n)$
Tree	$O(\log n)^*$ $O(n)^\dagger$	$O(n \log n)^*$ $O(n^2)^\dagger$	$O(\log n)^*$ $O(n)^\dagger$	$O(n)$
BalancedTree	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(n)$

* average case / † worst case

Part VII

Data Abstraction

ListAbs.hs (1)

```
module ListAbs
  (Set,nil,insert,set,element,equal,check) where
import Test.QuickCheck

data  Set a  =  MkSet [a]

nil :: Set a
nil =  MkSet []

insert :: a -> Set a -> Set a
insert x (MkSet xs)  =  MkSet (x:xs)

set :: [a] -> Set a
set xs  =  MkSet xs
```

ListAbs.hs (2)

```
element :: Eq a => a -> Set a -> Bool
x `element` (MkSet xs)  =  x `elem` xs

equal :: Eq a => Set a -> Set a -> Bool
MkSet xs `equal` MkSet ys  =
  xs `subset` ys && ys `subset` xs
  where
  xs `subset` ys  =  and [ x `elem` ys | x <- xs ]
```

ListAbs.hs (3)

```
prop_element :: [Int] -> Bool
prop_element ys  =
  and [ x `element` s == odd x | x <- ys ]
  where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_element

-- Prelude ListAbs> check
-- +++ OK, passed 100 tests.
```

ListAbsTest.hs

```
module ListAbsTest where
import ListAbs

test :: Int -> Bool
test n =
  s `equal` t
  where
  s = set [1,2..n]
  t = set [n,n-1..1]

-- Following no longer type checks!
-- breakAbstraction :: Set a -> a
-- breakAbstraction =  head
```

Hiding—the secret of abstraction

```
module ListAbs(Set,nil,insert,set,element,equal)

> ghci ListAbs.hs
Ok, modules loaded: SetList, MainList.
*ListAbs> let s0 = set [2,7,1,8,2,8]
*ListAbs> let MkSet xs = s0 in xs
Not in scope: data constructor ‘MkSet’
```

VS.

```
module ListUnhidden(Set(MkSet),nil,insert,element,equal)

> ghci ListUnhidden.hs
*ListUnhidden> let s0 = set [2,7,1,8,2,8]
*ListUnhidden> let MkSet xs = s0 in xs
[2,7,1,8,2,8]
*ListUnhidden> head xs
```

Hiding—the secret of abstraction

```
module TreeAbs(Set,nil,insert,set,element,equal)

> ghci TreeAbs.hs
Ok, modules loaded: SetList, MainList.
*TreeAbs> let s0 = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
Not in scope: data constructor ‘Node’, ‘Nil’
```

VS.

```
module TreeUnabs(Set(Node,Nil),nil,insert,element,equal)

> ghci TreeUnabs.hs
*SetList> let s0 = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
*SetList> invariant s0
False
```

Preserving the invariant

```
module TreeAbsInvariantTest where
import TreeAbs

prop_invariant_nil = invariant nil

prop_invariant_insert x s =
  invariant s ==> invariant (insert x s)

prop_invariant_set xs = invariant (set xs)

check =
  quickCheck prop_invariant_nil >>
  quickCheck prop_invariant_insert >>
  quickCheck prop_invariant_set

-- Prelude TreeAbsInvariantTest> check
-- +++ OK, passed 1 tests.
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```

It’s mine!