Informatics 1

Introduction to Computation

Lectures 12–13

Data Types and Data Abstraction

Philip Wadler

University of Edinburgh

Part I

Inf1A FP Midterm Feedback Survey

Inf1A FP Midterm Feedback Survey

https://www.surveymonkey.co.uk/r/YMYPRQN

Part II

2019 Inf1A FP Competition

2019 Inf1A FP Competition

- Prizes: Book vouchers. And glory!
- Number of prizes depend on number and quality of entries.
- Write a Haskell program with interesting graphics. Be creative!
- Previous year entries are online:

  `www.inf.ed.ac.uk/teaching/courses/inf1/fp/#competition`

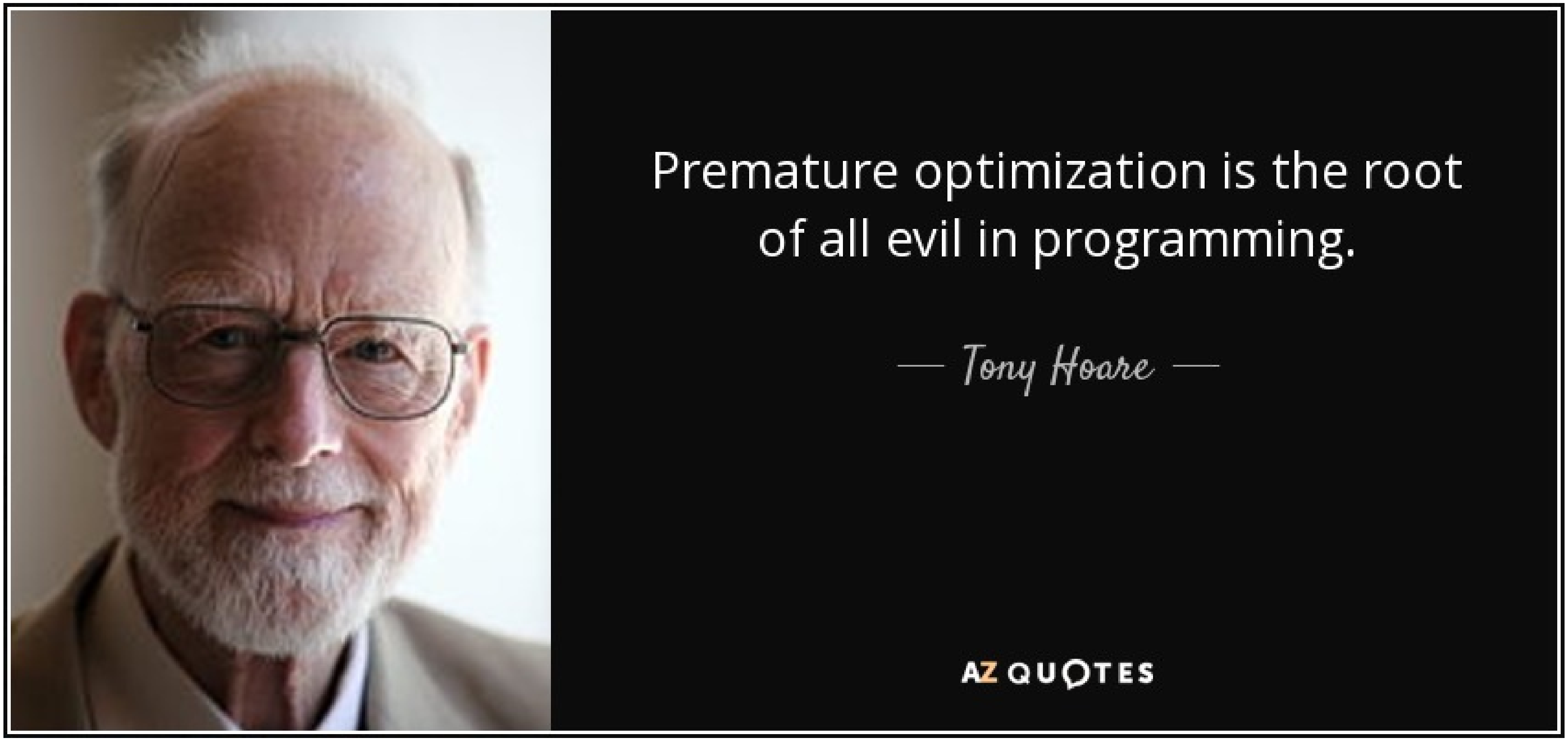
- Sponsored by Galois (`galois.com`)
- Submit code and image(s), list everyone who contributed.
- E-mail submissions
  to: Irene Vlassi-Pandi `<irene.vp@ed.ac.uk>` subject:
  `2019 Inf1A FP Competition`
- Submit by: **2pm Friday 15 November**

Part III

Efficiency and O-notation

Premature optimization is the root of all evil.

— Donald Knuth —


Premature optimization is the root of all evil in programming.
— Tony Hoare —
AZ QUOTES

Left vs. Right

Consider m lists, $xs_1, \ldots, xs_m$, each of length n.

Associated to the left, `foldl (++) []`

$$(((\,[] \mathbin{+\!\!+} xs_1) \mathbin{+\!\!+} xs_2) \mathbin{+\!\!+} xs_3) \cdots \mathbin{+\!\!+} xs_m$$

computing takes

$$\underbrace{0 + n + 2n + 3n + ... + (m-1)n}_{m \text{ times}}$$

steps. If we have m lists of length n, it takes $O(m^2 n)$ steps.

Associated to the right, `foldr (++) []`

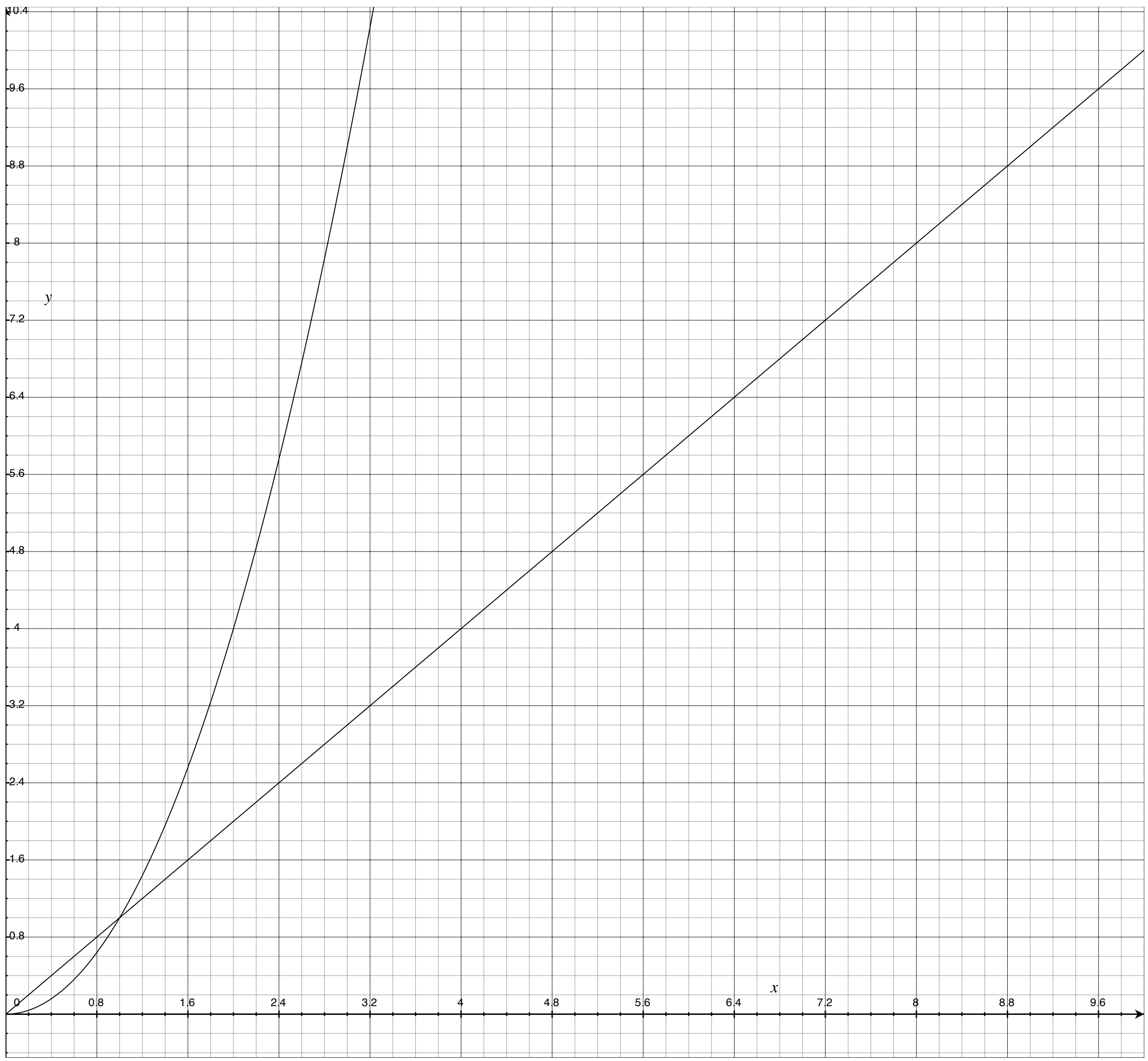
$$xs_1 \mathbin{+\!\!+} \cdots (xs_{m-2} \mathbin{+\!\!+} (xs_{m-1} \mathbin{+\!\!+} (xs_m \mathbin{+\!\!+} [\,])))$$

computing takes

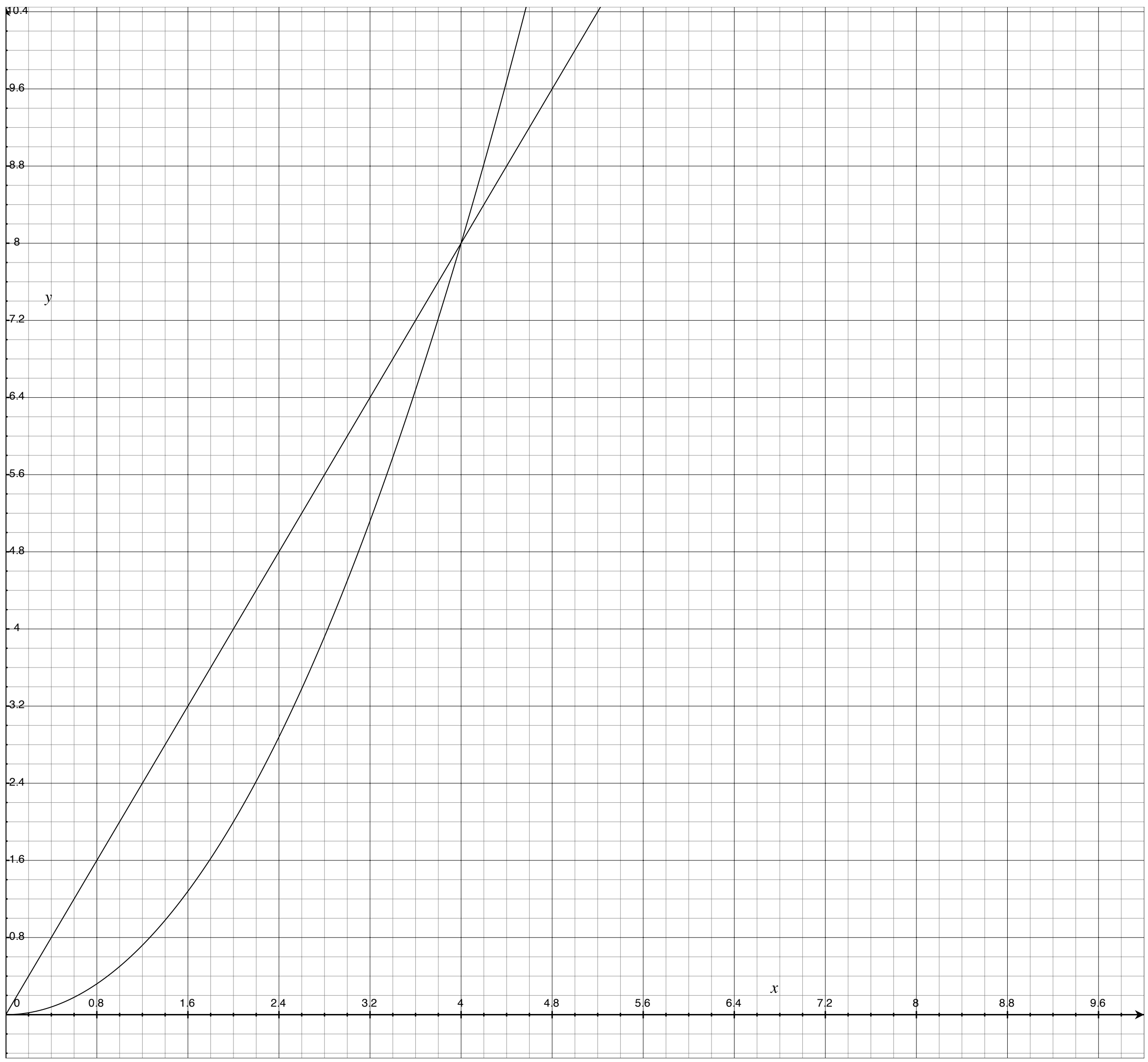
$$\underbrace{n + n + n + \cdots + n}_{m \text{ times}}$$

steps. If we have m lists of length n, it takes $O(mn)$ steps. When $m = 1000$, the first is a thousand times slower than the second!

$t = n$ 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
0
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
7.2
8
8.8
9.6
y
x

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



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

Big-O notation

Definition *We say* f *is* $O(g)$ *when* g *is an upper bound for* f*, for big enough inputs. To be precise,* f *is* $O(g)$ *if there are constants* c *and* m *such that* $f(n) \leq cg(n)$ *for all* $n \geq m$.

For instance: $2n + 10$ is $O(n)$ because $2n + 10 \leq 4n$ for all $n \geq 5$.

Big-O notation

Definition *We say f is $O(g)$ when g is an upper bound for f, for big enough inputs. To be precise, f is $O(g)$ if there are constants c and m such that $f(n) \leq cg(n)$ for all $n \geq m$.*

For instance: $2n + 10$ is $O(n)$ because $2n + 10 \leq 4n$ for all $n \geq 5$.

Constant factors don't matter

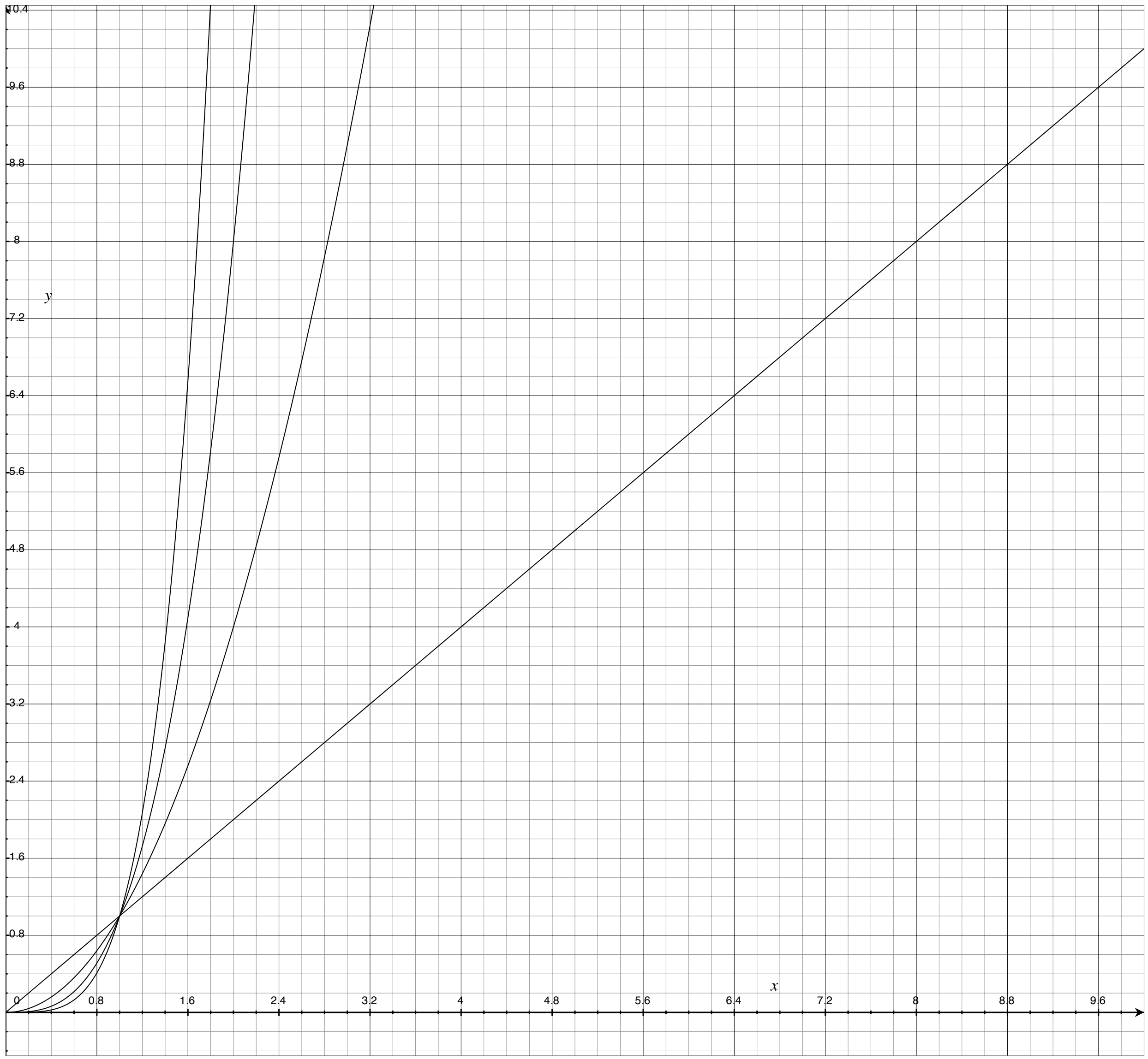
$O(n) = O(an + b)$, for any a and b

$O(n^2) = O(an^2 + bn + c)$, for any a, b, and c

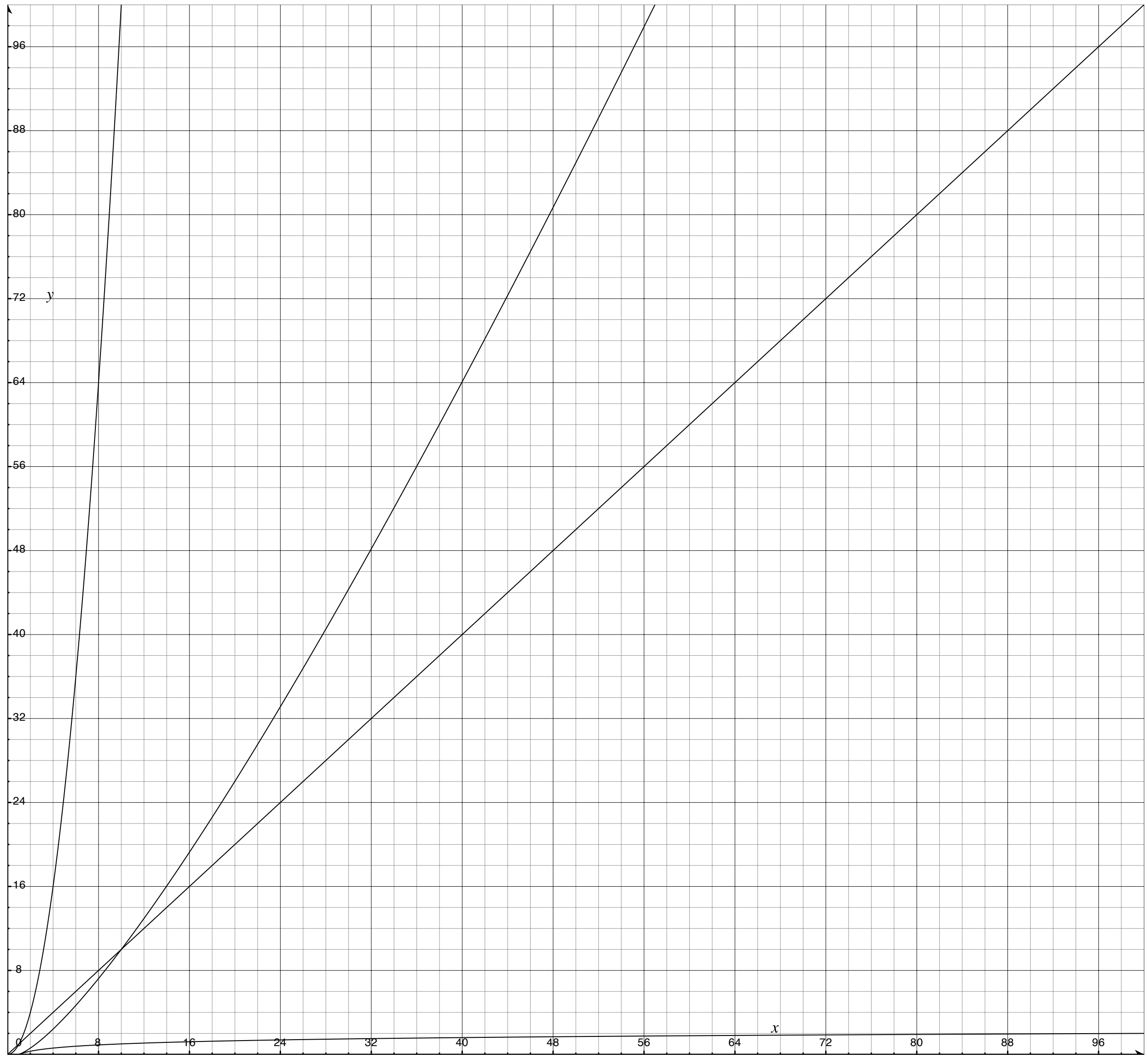
$O(n^3) = O(an^3 + bn^2 + cn + d)$, for any a, b, c, and d

$O(log_2(n)) = O(log_{10}(n))$

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



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



Associativity and Efficiency: Sequential vs. Parallel

Sequential:

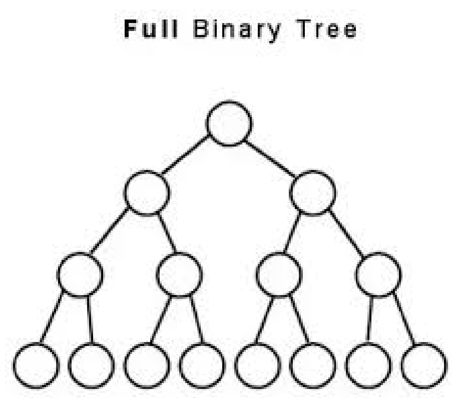
$$((((((x_1 + x_2) + x_3) + x_4) + x_5) + x_6) + x_7) + x_8$$

Summing 8 numbers takes 7 steps. If we have m numbers it takes $O(m)$ steps.

Parallel:

$$((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8))$$

Summing 8 numbers takes 3 steps.



If we have m numbers it takes $O(\log(m))$ steps. When $m = 1000$, the first is a hundred times slower than the second!

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

$O(\log n)$ “logarithmic”: parallel sum, divide and conquer search algorithms

$O(n)$ “linear”: ordinary sum

$O(n \log n)$: sorting algorithms

$O(2^n)$ “exponential”: tautology checking

Part IV

Sets as lists

List.hs (1)

```haskell
module List
  (Set,empty,insert,set,element,equal,check) where
import Test.QuickCheck

type Set a = [a]

empty :: Set a
empty =  []

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

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

List.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 ]
```

List.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 List> check
-- +++ OK, passed 100 tests.
```

Part V

Sets as *ordered* lists

OrderedList.hs (1)

```
module OrderedList
  (Set,empty,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) ]
```

OrderedList.hs (2)

```
empty :: Set a
empty =  []

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)
```

OrderedList.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
```

OrderedList.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 OrderedList> check
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Part VI

Sets as ordered trees

Tree.hs (1)

```
module Tree
  (Set(Nil,Node),empty,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 ]
```

Tree.hs (2)

```
empty :: Set a
empty  =  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 empty
```

Tree.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
```

Tree.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 Tree> check
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```

Part VII

Sets as *balanced* trees

BalancedTree.hs (1)

```haskell
module BalancedTree
  (Set(Nil,Node),empty,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
```

BalancedTree.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)
```

BalancedTree.hs (3)

```
empty :: Set a
empty  =  Nil

insert :: Ord a => a -> Set a -> Set a
insert x Nil  =  node empty x empty
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 empty
```

Part VIII

Sets as *balanced* trees without abstraction

BalancedTreeUnabs.hs (1)

```
module BalancedTreeUnabs
  (Set(Nil,Node),empty,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)

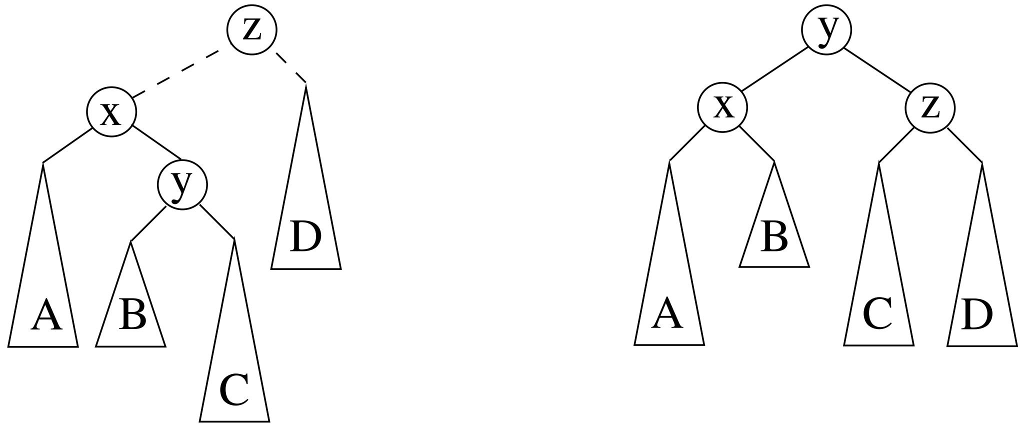
```
empty :: Set a
empty  =  Nil

insert :: Ord a => a -> Set a -> Set a
insert x Nil  =  node empty x empty
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 empty
```

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

```
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 IX

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 / $\dagger$ worst case

Part X

Data Abstraction

ListAbs.hs (1)

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

data  Set a  =  MkSet [a]

empty :: Set a
empty =  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,empty,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),empty,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,empty,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),empty,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

```haskell
module TreeAbsInvariantTest where
import TreeAbs

prop_invariant_empty = invariant empty

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

prop_invariant_set xs = invariant (set xs)

check =
  quickCheck prop_invariant_empty >>
  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!