Informatics 1

Functional Programming Lecture 19

Monday 1 December 2008

Complexity

Philip Wadler

University of Edinburgh

Part I

Append

How long does it take to append?

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys  =  ys
(x:xs) ++ ys  =  x:(xs ++ ys)

      "abcd" ++ "ef"
=   'a':("bcd" ++ "ef")
=   'a':('b':("cd" ++ "ef"))
=   'a':('b':('c':("d" ++ "ef")))
=   'a':('b':('c':('d':("" ++ "ef"))))
=   'a':('b':('c':('d':"ef")))
=   "abcdef"
```

Computing `xs++ys` takes time proportional to the length of `xs`.

Linear vs. quadratic append

Associate to the right

```
"a"++("b"++("c"++("d"++("e"++[]))))   =   "abcde"
```

$$\underbrace{1 + \cdots + 1}_{n \text{ times}} = n$$

Associate to the left

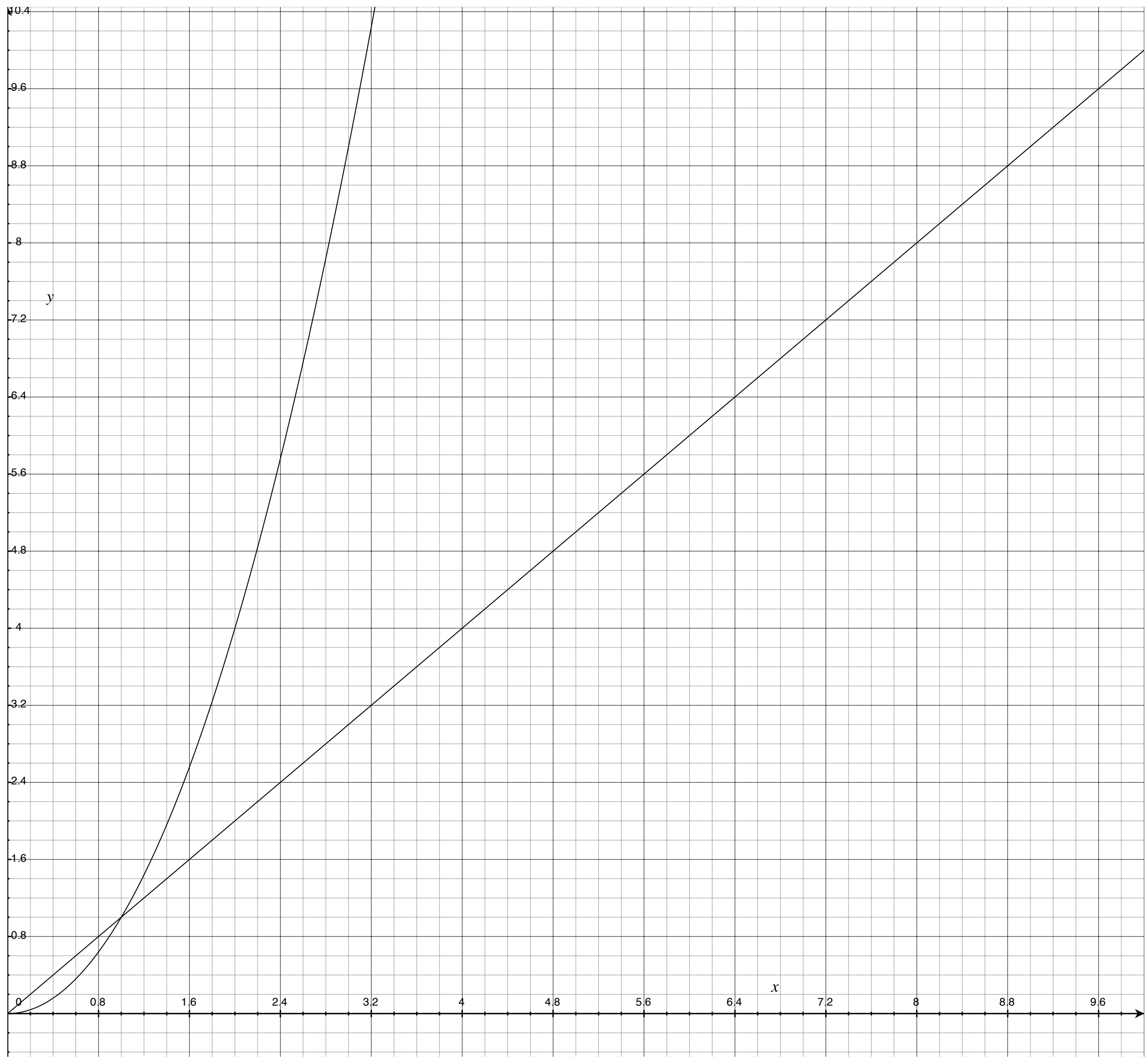
```
(((([]++"a")++"b")++"c")++"d")++"e")   =   "abcde"
```

$$\underbrace{0 + 1 + \cdots + (n-1)}_{n \text{ times}} = \frac{n(n-1)}{2}$$

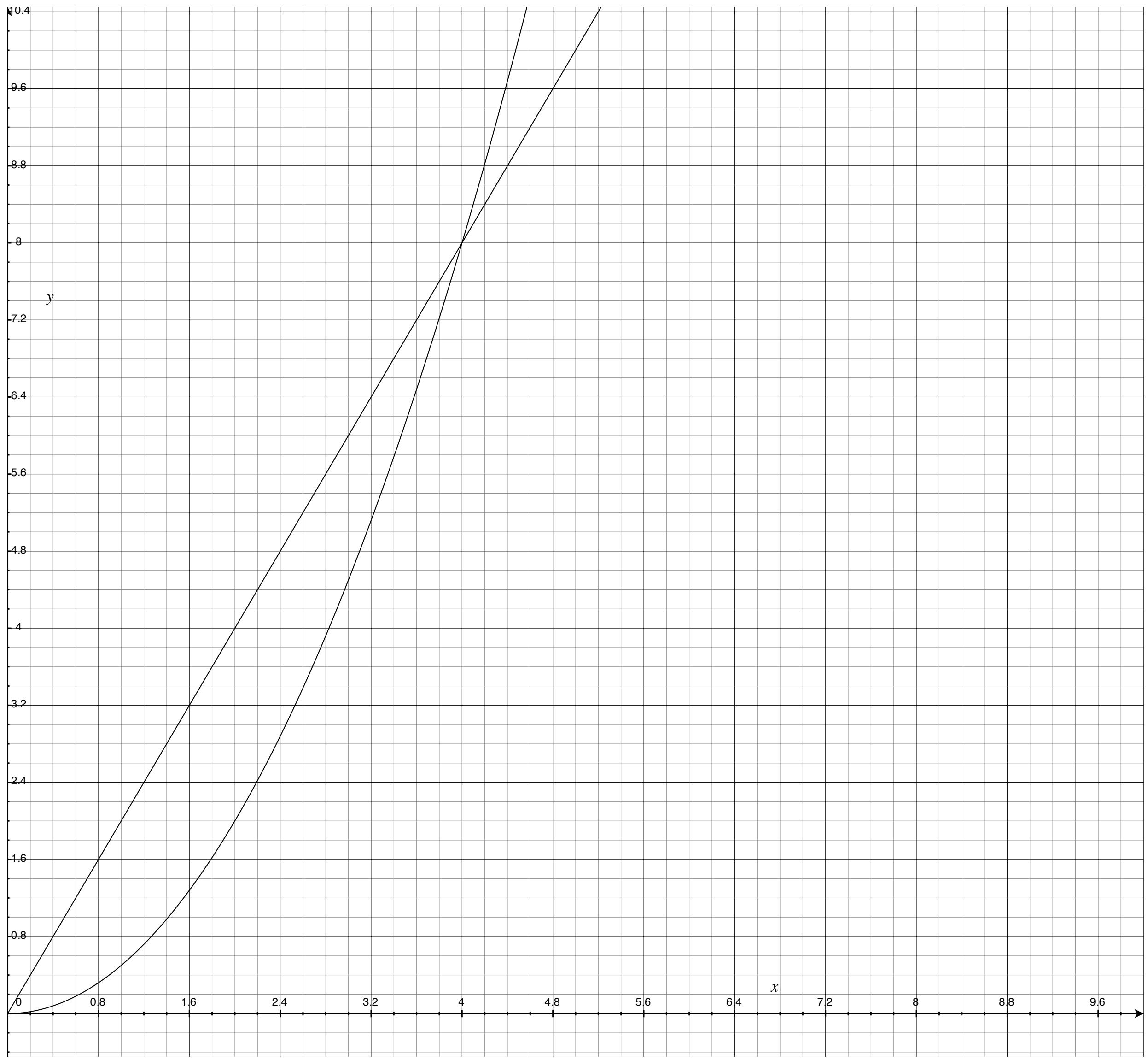
Part II

Complexity

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

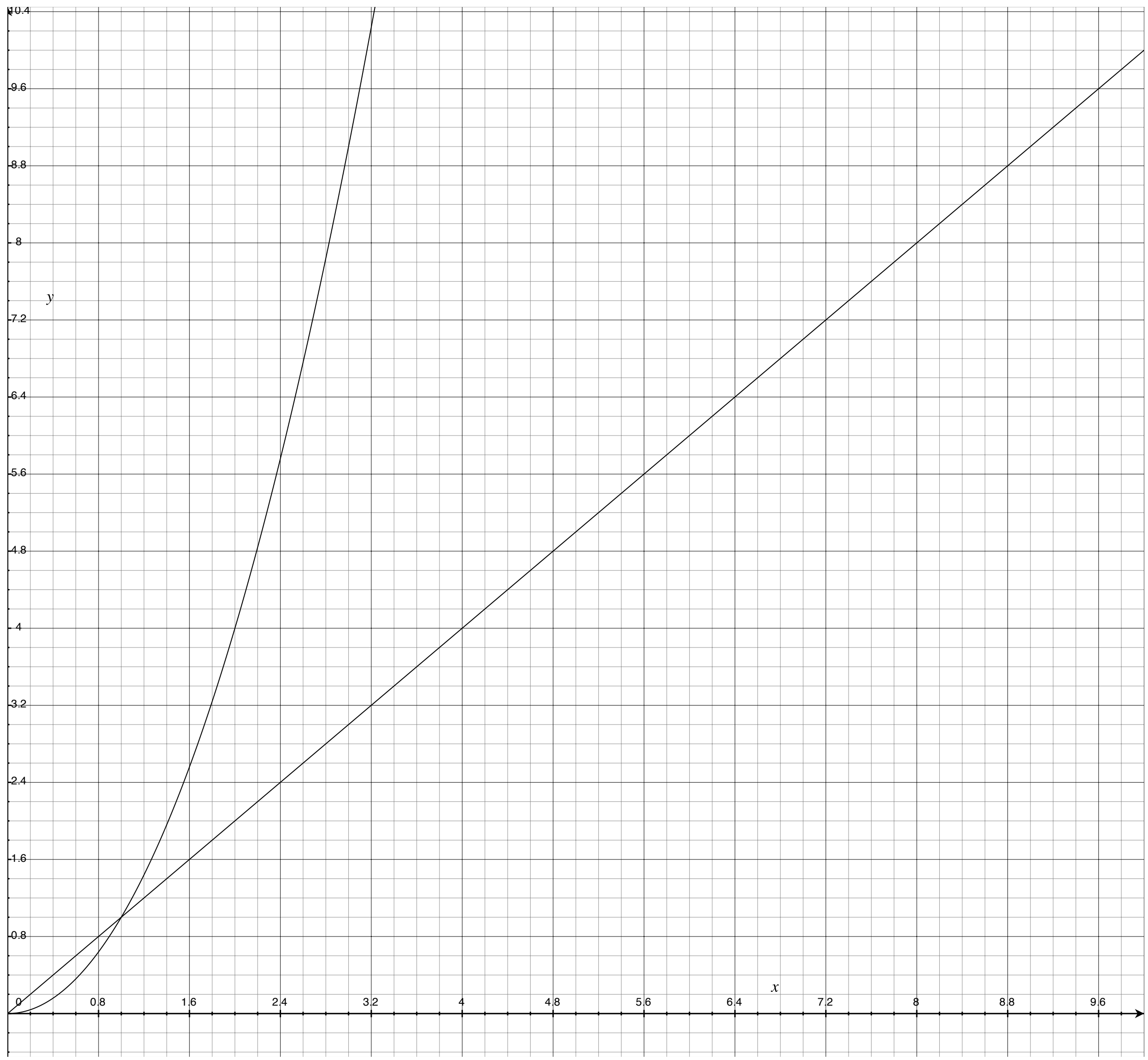

10.4
9.6
8.8
8
y
7.2
6.4
5.6
4.8
4
3.2
2.4
1.6
0.8
x
0
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
7.2
8
8.8
9.6

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



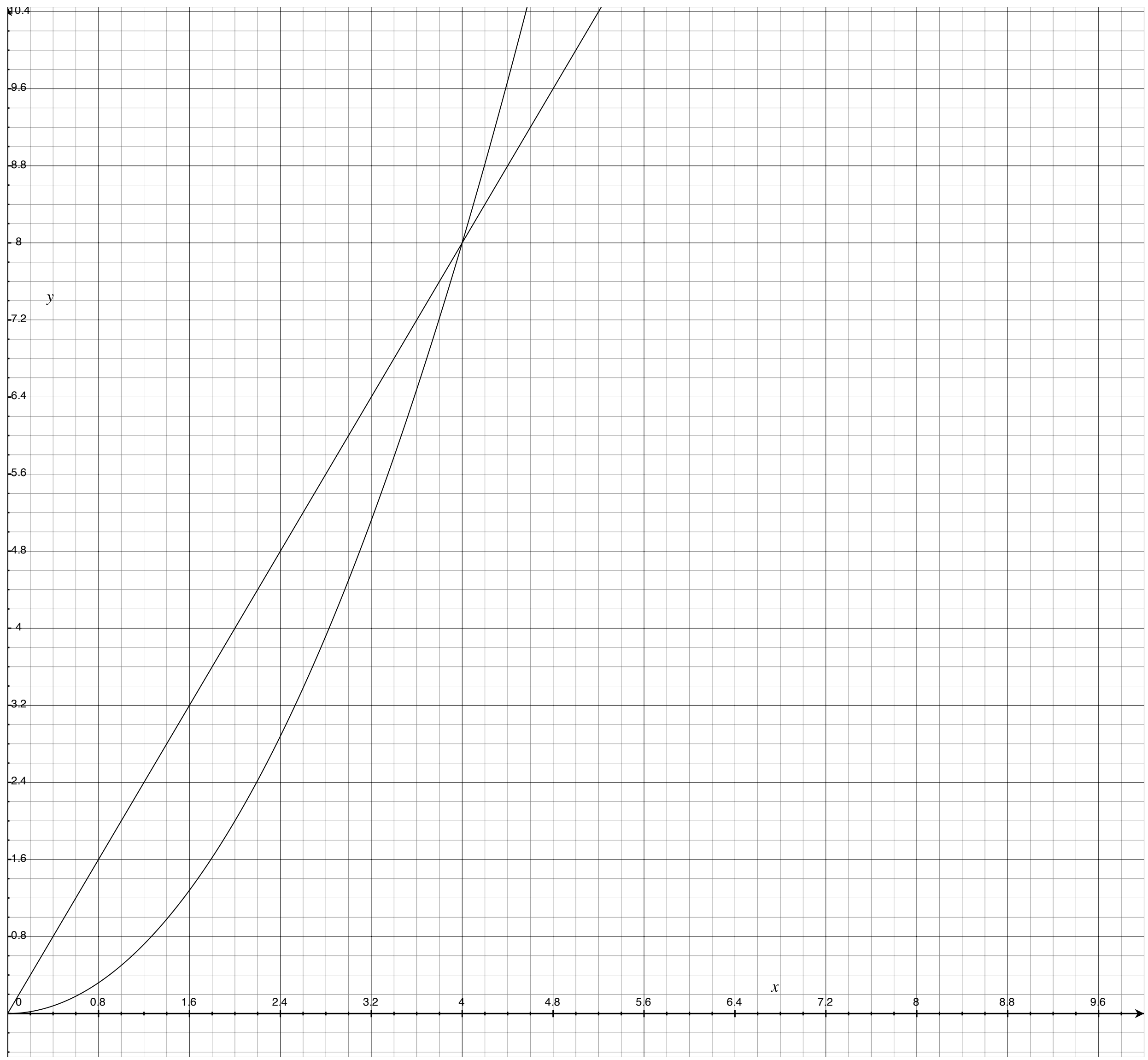
10.4
9.6
8.8
8
y
7.2
6.4
5.6
4.8
4
3.2
2.4
1.6
0.8
x
0
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
7.2
8
8.8
9.6

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

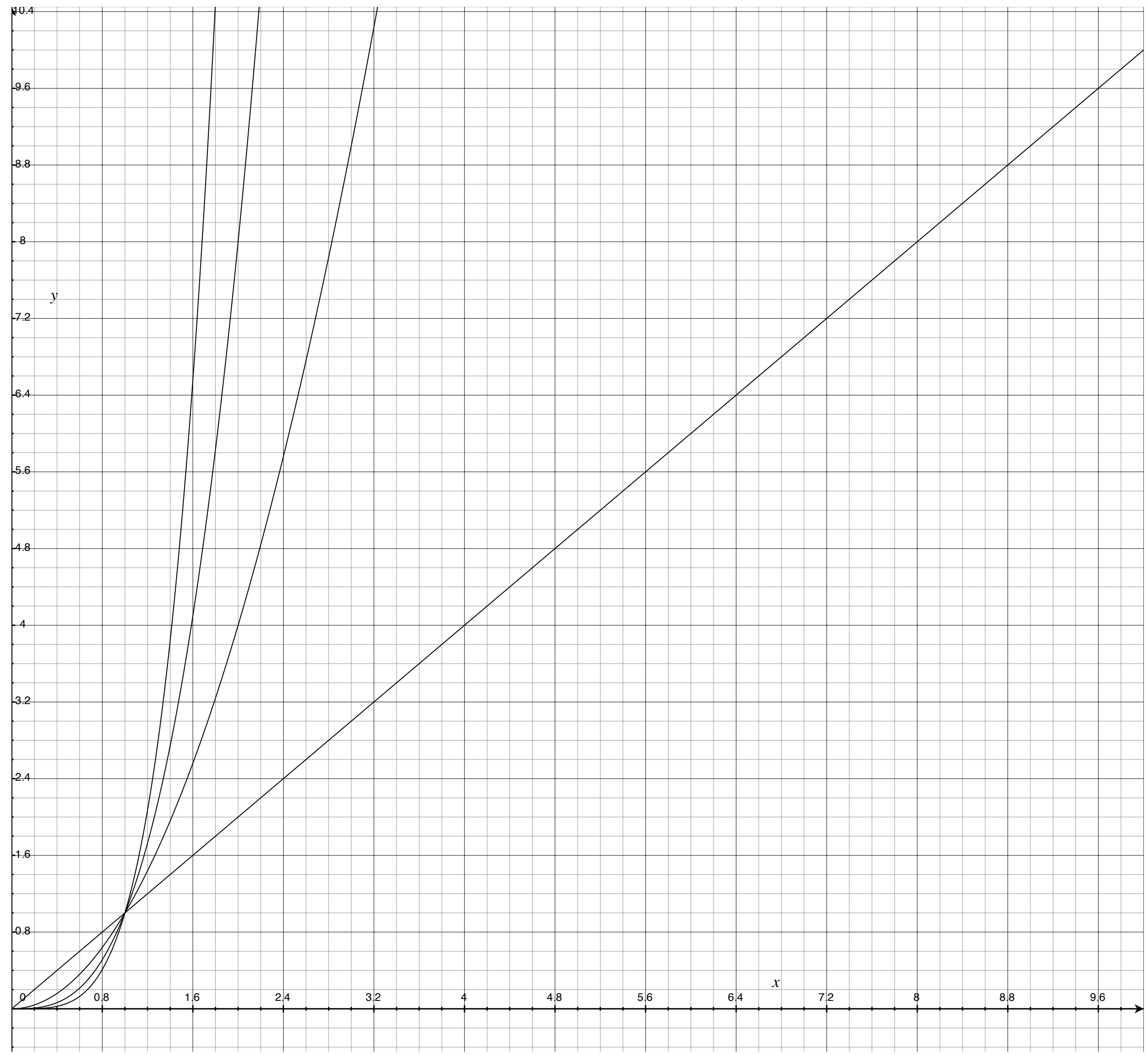


10.4
9.6
8.8
8
y
7.2
6.4
5.6
4.8
4
3.2
2.4
1.6
0.8
0
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
x
7.2
8
8.8
9.6

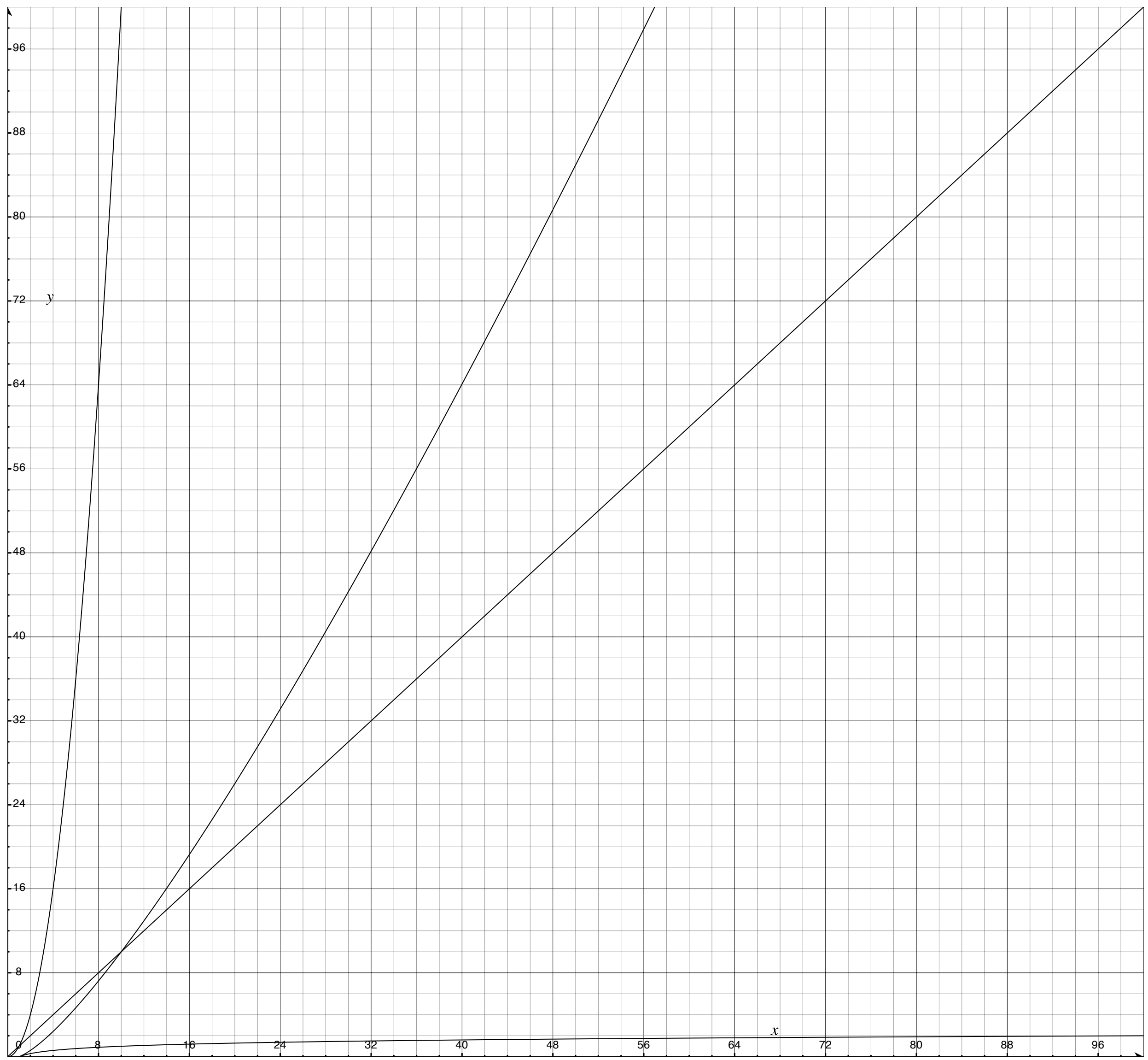
$O(n)$ vs $O(n^2)$


y
x
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

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



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



Part III

Fold right and fold left

Fold right and fold left

```
foldr (+) 0 [1,2,3,4]  =  1+(2+(3+(4+0)))
foldl (+) 0 [1,2,3,4]  =  (((0+1)+2)+3)+4


foldr :: (x -> a -> a) -> a -> [x] -> a
foldr f a []       =  a
foldr f a (x:xs)   =  f x (foldr f a xs)

foldl :: (a -> x -> a) -> a -> [x] -> a
foldl f a []       =  a
foldl f a (x:xs)   =  (foldl f (f a x) xs)
```

Fold right, sum

```
    foldr (+) 0 [1..4]
  =
    1 + foldr (+) 0 [2..4]
  =
    1 + (2 + foldr (+) 0 [3..4])
  =
    1 + (2 + (3 + foldr (+) 0 [4..4]))
  =
    1 + (2 + (3 + (4 + foldr (+) 0 [5..4])))
  =
    1 + (2 + (3 + (4 + foldr (+) 0 [])))
  =
    1 + (2 + (3 + (4 + 0)))
  =
    10
```

Linear time, linear space

Fold right, sum

```
  foldl (+) 0 [1..4]
=
  foldr (+) (0+1) [2..4]
=
  foldr (+) ((0+1)+2) [3..4]
=
  foldr (+) ((0+1)+2)+3) [4..4]
=
  foldr (+) (((0+1)+2)+3)+4) [5..4]
=
  foldr (+) (((0+1)+2)+3)+4) []
=
  (((0+1)+2)+3)+4)
=
  10
```

But this does not reflect the space behaviour!

Fold left, sum

```
  foldl (+) 0 [1..4]
=
  foldl (+) (0+1) [2..4]
=
  foldl (+) 1 [2..4]
=
  foldl (+) (1+2) [3..4]
=
  foldl (+) 3 [3..4]
=
  foldl (+) (3+3) [4..4]
=
  foldl (+) 6 [4..4]
=
  foldl (+) (6+4) [5..4]
=
  foldl (+) 10 []
=
  10
```

Linear time, constant space

Fold right, append

```
   foldr (++) [] ["a","b","c","d"]
=
   "a" ++ foldr (++) [] ["b","c","d"]
=
   "a" ++ ("b" ++ foldr (++) [] ["c","d"])
=
   "a" ++ ("b" ++ ("c" ++ foldr (++) [] ["d"]))
=
   "a" ++ ("b" ++ ("c" ++ ("d" ++ foldr (++) [] [])))
=
   "a" ++ ("b" ++ ("c" ++ ("d" ++ [])))
=
   "abcd"
```

Linear time, linear space

Fold left, append

```
  foldl (++) [] ["a","b","c","d"]
=
  foldl (++) ([] ++ "a") ["b","c","d"]
=
  foldl (++) "a" ["b","c","d"]
=
  foldl (++) ("a" ++ "b") ["c","d"]
=
  foldl (++) "ab" ["c","d"]
=
  foldl (++) ("ab" ++ "c") ["d"]
=
  foldl (++) "abc" ["d"]
=
  foldl (++) ("abc" ++ "d") []
=
  foldl (+) "abcd" []
=
  "abcd"
```

Linear time, constant space

Part IV

Sort

Insert

```
insert :: Int -> [Int] -> [Int]
insert x []                          =  [x]
insert x (y:ys) | x > y              =  y : insert x ys
                | otherwise          =  x : y : ys
```

```
  insert 3 [1,2,4]
=
  insert 3 (1:2:4:[])
=
  1 : insert 3 (2:4:[])
=
  1 : 2 : insert 3 (4:[])
=
  1 : 2 : 3 : 4 : []
=
  [1,2,3,4]
```

Insertion sort

```
isort :: [Int] -> [Int]
isort []       =  []
isort (x:xs)   =  insert x (isort xs)

  isort [3,1,4,2]
=
  insert 3 (isort [1,4,2])
=
  insert 3 (insert 1 (isort [4,2]))
=
  insert 3 (insert 1 (insert 4 (isort [2])))
=
  insert 3 (insert 1 (insert 4 (insert 2 [])))
=
  insert 3 (insert 1 (insert 4 [2]))
=
  insert 3 (insert 1 [2,4])
=
  insert 3 [1,2,4]
=
  [1,2,3,4]
```

Insertion sort

```
insert :: Ord a => a -> [a] -> [a]
insert x []                    =  [x]
insert x (y:ys) | x > y        =  y : insert x ys
                | otherwise    =  x : y : ys
```

constant time $O(1)$, best case

linear time $O(n)$, average case

```
isort :: Ord a => [a] -> [a]
isort []       =  []
isort (x:xs)   =  insert x (isort xs)
```

linear time $O(n)$, best case

quadratic time $O(n^2)$, average case

Insertion sort, higher order

```
insert :: Ord a => a -> [a] -> [a]
insert x ys  =
  takeWhile (x>) ys ++ [x] ++ dropWhile (x>) ys

isort :: Ord a => [a] -> [a]
isort xs  =  foldr insert [] xs
```

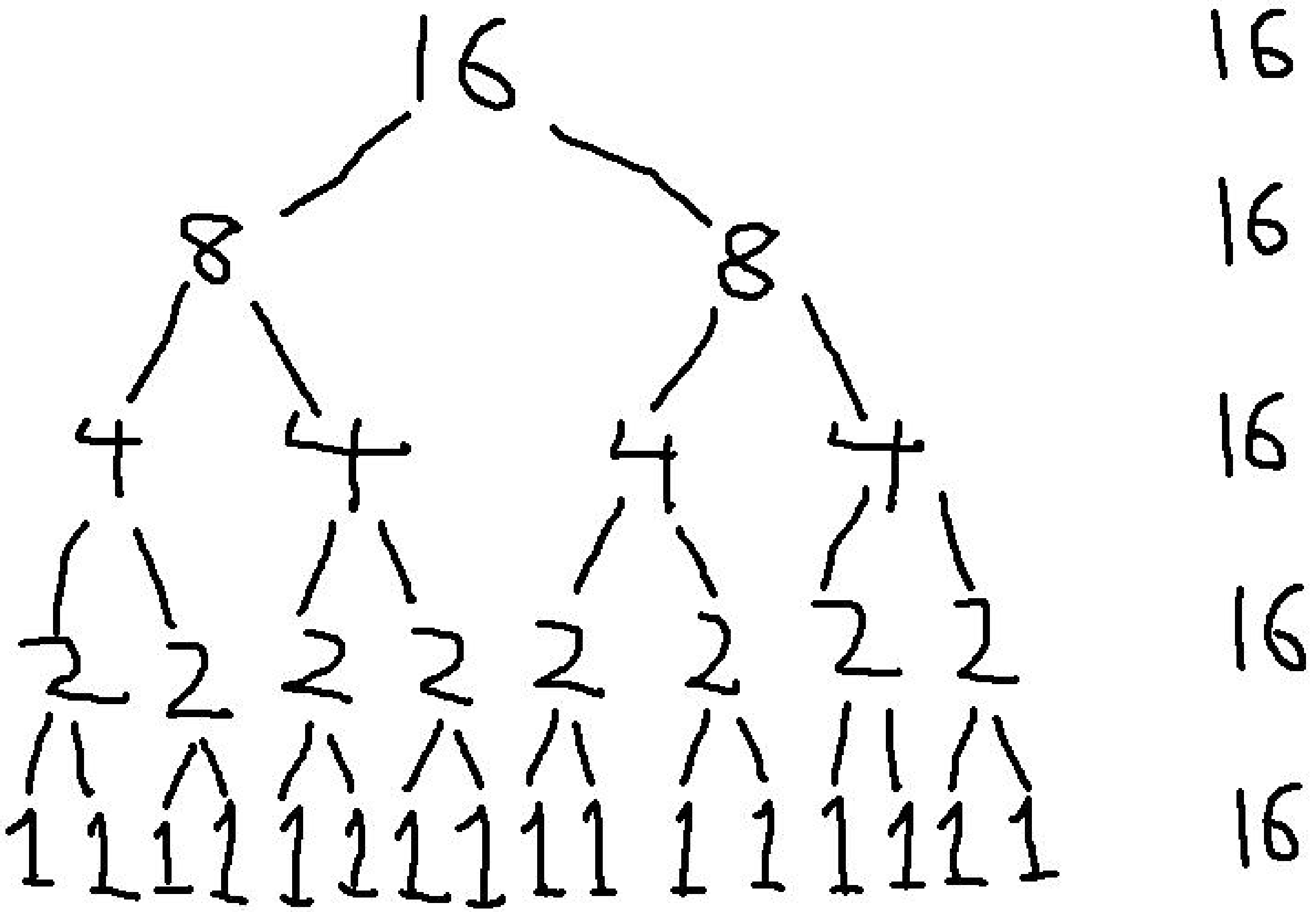
Quicksort

```
qsort :: Ord a => [a] -> [a]
qsort []        =  []
qsort (x:ys)    =
  qsort [ y | y <- ys, y < x ] ++
  [x] ++
  qsort [ y | y <- ys, y >= x ]
```

$O(n \log n)$, average case

$O(n^2)$, worst case

Where the logs come from



Merge sort

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys             =  ys
merge (x:xs) []         =  x:xs
merge (x:xs) (y:ys)
   | x < y              =  x : merge xs (y:ys)
   | otherwise          =  y : merge (x:xs) ys

split :: [a] -> ([a],[a])
split []         =  ([], [])
split [x]        =  ([x], [])
split (x:y:zs)   =  (x:xs, y:ys)
   where
   (xs,ys) = split zs

msort :: Ord a => [a] -> [a]
msort []    =  []
msort [x]   =  [x]
msort zs    =  merge (msort xs) (msort ys)
   where
   (xs,ys) = split zs
```

Merge sort

$O(n \log n)$, always