

Informatics 1
Introduction to Computation
Lecture 14

Laziness, Higher-order, and Sorting

Philip Wadler
University of Edinburgh

Part I

The importance of being lazy

Searching for the first odd number

```
ho :: Int -> [Int]
ho n  =  (take 1 . filter odd) [0..n]

comp :: Int -> [Int]
comp n  =  take 1 [ x | x <- [0..n], odd x ]

rec :: Int -> Int
rec n  =  helper 0
  where
    helper :: Int -> [Int]
    helper i | i > n      =  []
              | odd i     =  [i]
              | otherwise =  helper (i+1)
```

Quickcheck

```
prop_odd :: Int -> Bool
prop_odd n = a == b && b == c
  where
    a = ho n
    b = comp n
    c = rec n
```

```
[1 of 1] Compiling Main
```

```
Ok, one module loaded.
```

```
*Main> quickCheck prop_odd
```

```
+++ OK, passed 100 tests.
```

Timing

```
*Main> :set +s
*Main> ho 1000000
[1]
(0.00 secs, 64,776 bytes)
*Main> comp 1000000
[1]
(0.00 secs, 64,984 bytes)
*Main> rec 1000000
[1]
(0.00 secs, 65,168 bytes)
```

How it works: rec

```
rec :: Int -> Int
rec n = helper 0
  where
    helper :: Int -> [Int]
    helper i | i > n      = []
              | odd i      = [i]
              | otherwise = helper (i+1)
```

```
rec 1000000
=
  helper 0
=
  helper 1
=
  [1]
```

How it works: ho

```
ho :: Int -> [Int]
ho n = (take 1 . filter odd) [0..n]
```

```
ho 1000000
=
  (take 1 . filter odd) [0..1000000]
=
  take 1 (filter odd [0..1000000])
=
  take 1 (filter odd (0 : [1..1000000]))
=
  take 1 (filter odd ([1..1000000]))
=
  take 1 (filter odd (1 : [2..1000000]))
=
  take 1 (1 : filter odd [2..1000000])
=
  1 : take 0 (filter odd [2..1000000])
```

Part II

Sum of odd squares
three ways

Sum of odd squares

```
ho :: Int -> Int
ho n = (foldl (+) 0 . map (^2) . filter odd) [0..n]
```

```
comp :: Int -> Int
comp n = sum [ x^2 | x <- [0..n], odd x ]
```

```
rec :: Int -> Int
rec n = helper 0 0
  where
    helper :: Int -> Int -> Int
    helper i a | i > n      = a
               | odd i      = helper (i+1) (a + i^2)
               | otherwise  = helper (i+1) a
```

Quickcheck

```
prop_sqr :: Int -> Bool
prop_sqr n = a == b && b == c
  where
    a = ho n
    b = comp n
    c = rec n
```

Ok, one module loaded.

```
*Main> quickCheck prop_sqr
```

```
+++ OK, passed 100 tests.
```

Runtimes in ghci

```
*Main> :set +s
*Main> ho 1000000
166666666666500000
(0.43 secs, 596,687,792 bytes)
*Main> comp 1000000
166666666666500000
(0.67 secs, 628,685,832 bytes)
*Main> rec 1000000
166666666666500000
(1.02 secs, 692,881,968 bytes)
```

The Moral

Usually coding involves tradeoffs:

simple and *slow*

vs.

complex and *fast*.

The big win is when you can apply maths to be
both *simple* and *fast*.

Part III

Sorting three ways

Insertion sort

```
isort :: Ord a => [a] -> [a]
```

```
isort = foldr insert []
```

where

```
insert :: Ord a => a -> [a] -> [a]
```

```
insert x [] = [x]
```

```
insert x (y : ys) | x <= y = x : y : ys  
                  | otherwise = y : insert x ys
```

Quicksort

```
qsort :: Ord a => Int -> [a] -> [a]
qsort k xs | length xs <= k = isort xs
qsort k (y:xs) =
  qsort k [ x | x <- xs, x < y ]
  ++ [ y ] ++
  qsort k [ x | x <- xs, x >= y ]
```

Merge sort

```
msort :: Ord a => Int -> [a] -> [a]
msort k xs | length xs <= k = isort xs
           | otherwise      = merge (msort k (take m xs))
                                   (msort k (drop m xs))
```

where

```
m = length xs `div` 2
```

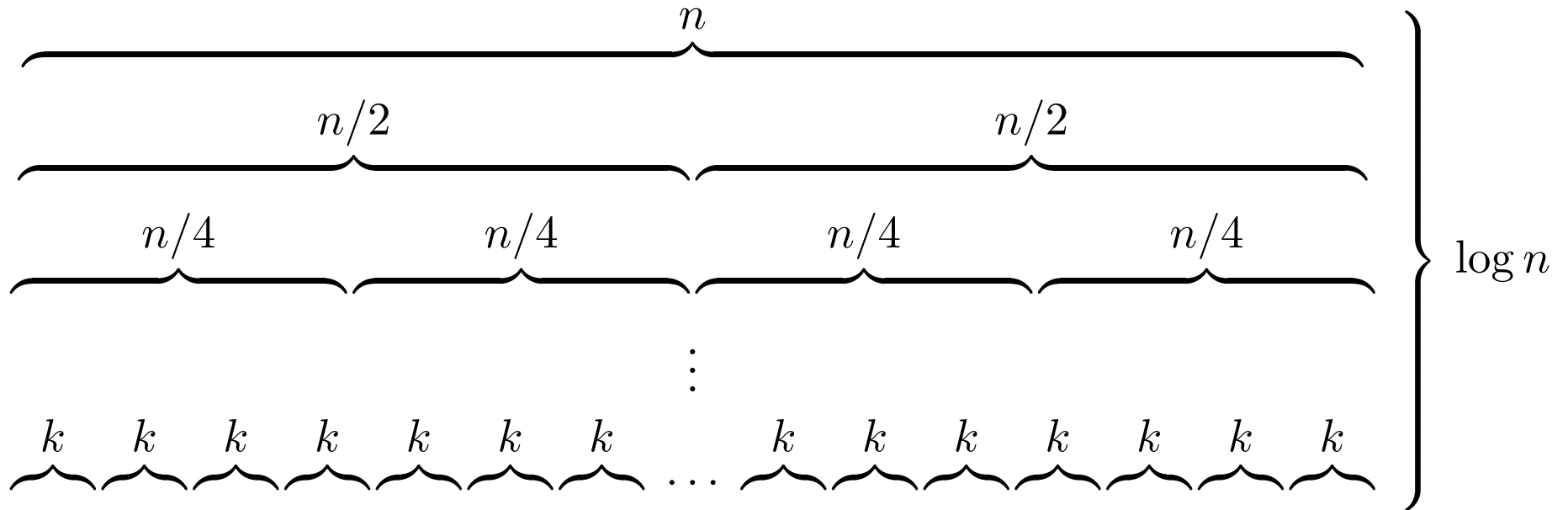
```
merge :: Ord a => [a] -> [a] -> [a]
```

```
merge xs [] = xs
```

```
merge [] ys = ys
```

```
merge (x:xs) (y:ys) | x <= y = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys
```


Why quicksort and mergesort are $O(n \log n)$



n number of elements to be sorted

k cutoff size

Part IV

A few graphs

