Review
Higher-Order Functions
Trees

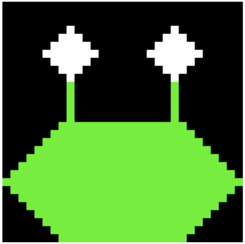
January 30–31, 2012
CS 60: Principles of Computer Science

Assignment 1 due Monday 1/30
Assignment 2 due Monday 2/6

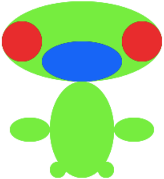
Aliens!

> (alien 20 "darkgreen")

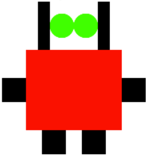
> (alien 10)



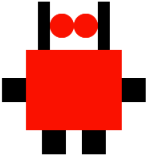
> (alien 10)



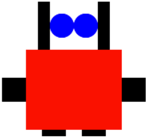
> (alien "green")



> (alien "red")



> (alien "blue")



In the world of Big-O...

- ✓ *Constant* factors are ignored
- ✓ Inputs are arbitrarily large (so "small" summands can be ignored)
- ✓ We are looking for upper bounds.

$O(1)$:
- 6 steps
- 1 (big?) step
- no more than 4000 steps
- somewhere between 2 and 47 steps, depending on the input.

$O(n)$:
- $100n + 3$ steps
- $n/2 + 7\log_2 n + 10,000,000$ steps
- anywhere between 3 and 69 steps per item, for n items.

$O(n^2)$:
- $2n^2 + 100n + 3$ steps
- $n^2/3$ steps
- somewhere between 1 and 40 steps per item, for n^2 items
- anywhere between 1 and $7n$ steps per item, for n items.

Computing Length

```
; Given a list L, return its length.
; Already built-in as the function "length"
 (define (myLength L)
   (cond
       [ (null? L)  0 ]
       [ else       (+ 1 (myLength (rest L))) ]
   ))
```

Running time?

Let n be the length of the initial list and let $T(n)$ be the (big-O) steps for an input of size n.

- ✓ Then $T(0) = 1$; otherwise $T(n) = 1 + T(n-1)$.
- ✓ So, $T(n) = 1 + T(n-1) = 2 + T(n-2) = \cdots = n + T(0) = O(n)$.

Alternatively:

- ✓ We look at n elements of the list
- ✓ We do $O(1)$ work per item (testing for empty, adding one, etc.)
- ✓ $O(n) \times O(1) = O(n \times 1) = O(n)$.

Append

```
; Already built-in, but this is how it works.
(define (append L M)
  (cond
     [ (null? L)  M ]
     [ else  (cons (first L) (append (rest L) M)) ]
   ))
```

Running time?

Let n be the length of the *first* list and let $T(n)$ be the (big-O) steps for an input of size n.

- ✓ Then (asymptotically) $T(0) = 1$; otherwise $T(n) = 1 + T(n-1)$.
- ✓ So, $T(n) = 1 + T(n-1) = 2 + T(n-2) = \cdots = n + T(0) = O(n)$.

Reverse

How efficient is it?

```
(define (rev L)
 (cond
   [ (null? L)  '() ]
   [  else      (append (rev (rest L))
                        (list (first L)))  ] ))
```

Let n be the length of the input and let $T(n)$ be the (big-O) steps for an input of size n.

- ✓ Then (asymptotically!) $T(0) = 1$; otherwise $T(n) = n + T(n-1)$.
- ✓ So, $T(n) = n + T(n-1) = n + (n-1) + T(n-2) = \cdots = n + (n-1) + \cdots + 3 + 2 + 1 + T(0) = O(n^2)$.

Comparing (fac 4)

General Recursion: $(4 \times (3 \times (2 \times 1)))$

```
(fac 4)  =  (4 * (fac 3))
         =  (4 * (3 * (fac 2)))
         =  (4 * (3 * (2 * (fac 1))))
         =  (4 * (3 * (2 * 1)))
         =  (4 * (3 * 2))
         =  (4 * 6)
         =  24
```

Tail Recursion: $(((4 \times 3) \times 2) \times 1)$

```
(fac 4)  =  (fac-help 4 1)
         =  (fac-help 3 4)
         =  (fac-help 2 12)
         =  (fac-help 1 24)
         =  24
```

Two Approaches To Using Recursion

"General Recursion"

- ✓ Take in a number (or list)
- ✓ Recurse on precessor (or `rest`)
- ✓ Compute answer for the given input

```
(define (fac N)
 (cond
  [ (< N 2)  1 ]
  [ else (* N (fac (- N 1))) ]
 )
)
```

"Tail Recursion"

- ✓ Add an extra "accumulator" argument
- ✓ The accumulator summarizes the numbers/list elements seen so far
- ✓ At the end, return the accumulator.

```
(define (fac N)   (fac-help N 1)

(define (fac-help N A)
 (cond
  [ (< N 2)  A ]
  [ else (fac-help (- N 1) (* N A))
 )
)
```

Comparing `(rev '(1 2 3 4))`

General Recursion

```
(rev '(1 2 3 4))
  =   (append (rev '(2 3 4)) '(1))
  =   (append (append (rev '(3 4)) '(2)) '(1))
  =   (append (append (append (rev '(4)) '(3)) '(2)) '(1))
  =   (append (append (append (append (rev '()) '(4)) '(3)) '(2)) '(1))
  =   (append (append (append (append '() '(4)) '(3)) '(2)) '(1))
  =   (append (append (append '(4) '(3)) '(2)) '(1))
  =   (append (append '(4 3) '(2)) '(1))
  =   (append '(4 3 2) '(1))
  =   '(4 3 2 1)
```

Tail Recursion

```
(rev '(1 2 3 4))   =   (rev-help '(1 2 3 4) '())
                   =   (rev-help '(2 3 4) '(1) )
                   =   (rev-help '(3 4) '(2 1) )
                   =   (rev-help '(4) '(3 2 1) )
                   =   (rev-help '() '(4 3 2 1) )
                   =   '(4 3 2 1)
```

Tail-Recursive Reverse

```
(define (rev L)
  (rev-help L '()))

;; L contains the remaining elements to reverse
;; A contains the reverse of the elements we've seen so far
(define (rev-help L A)
 (cond
   [ (null? L)  A ]
   [  else (rev-help (rest L) (cons (first L) A)) ]
 )
)
```

How efficient is it?

Since `cons`, `first`, `rest`, *etc.* are all $O(1)$, for a list of length n we can define the number of steps $T(n)$ needed for reverse by $T(0) = 1$ and $T(n) = 1 + T(n-1)$. Thus, $O(n)$.

Does this work?

```
(define (fast-pow b p)
  (cond
    ((< p 1) 1)  ; base case

    ((odd? p)
       (* b (fast-pow b (- p 1)) ))  ; odd case

    ( else
       (* (fast-pow b (/ p 2))
          (fast-pow b (/ p 2)))) ))  ; even case
```

It produces the right answer, but it unnecessarily repeats work. For example, `(fastpow b 16)` computes `(fastpow b 8)` twice, which together computes `(fastpow b 4)` four times, `(fastpow b 2)` eight times, ... One can show that the running time is $O(n)$.

let* IT BE $O(\log n)$

```
(define (fast-pow b p)
  (cond
    ((< p 1) 1)  ; base case

    ((odd? p)
       (* b (fast-pow b (- p 1)) ))  ; odd case

    ( else

       (let* ( [ halfp (/ p 2)             ]
               [ bp2   (fast-pow b halfp) ] )
             (* bp2 bp2)                 ; even case
       )
    )
```

Good News, Bad News

Recursion is very common in Racket, but you can often avoid it completely!
How are these similar? How are these different?

```
;; find squares of numbers
(define (squares L)
  (if (equal? L '())
    '()
     (cons (square (first L))
           (squares (rest L)))
  )
)

;; (squares '(1 2 3 4))
;;          ===> '(1 4 9 16)
```

```
;; find factorials of numbers
(define (facs L)
  (if (equal? L '())
    '()
    (cons (fac (first L))
          (facs (rest L)))
  )
)

;; (facs '(1 2 3 4))
;;        ===> '(1 2 6 24)
```

map: A Recursion Alternative

```
;; apply function f to all the elements in L
(define (map f L)
  (if (null? L)
      '()
      (cons (f (first L)) (map f (rest L)))))

(define (facs L) (map fac L))

(define (squares L)
```

Some "Anonymous Functions" in Racket

- ✓ The "successor function" (Does the name x matter?)

```
(lambda (x) (+ x 1))
```

- ✓ The "geometric mean" function

```
(lambda (x y) (sqrt (* x y)))
```

- ✓ The "is-greater-than-5 function"

```
(lambda (N) (> N 5))
```

- ✓ The "is-a-list-of-length-two function"

```
(lambda (L) (and (list? L) (= (len L) 2)))
```

- ✓ The squaring function?

Syntactic Sugar Makes Racket Sweeter

The function definitions we saw earlier:

```
(define (square x)  (* x x))
```

are just abbreviations for:

```
(define square  (lambda (x) (* x x)))
```

Of course, you don't *have* to give functions names before using them:

```
( (lambda (x) (+ x x))  12 )
```

Higher-Order Functions

Functions that take functions as arguments (like `map`) or return functions as results…

```
(define (wrap tag)
  (lambda (text) (list tag text tag)))
```

What is `( (wrap 'w) 'o))` ?

1. Error
2. `#procedure`
3. '(*w o*)
4. '(*w o w*)
5. '(*o w o*)

What does foldr do?

```
(foldr  +  0  '(3 4 5))     ;; ===>   12

(foldr cons '() '(1 2 3 4)) ;; ===>  '(1 2 3 4)
```

Notes:

- ✓ Sometimes called `reduce`
- ✓ There's also a `foldl`

What do filter and sort do?

```
(filter odd? '(1 2 3 4 5))     ;; ===> '(1 3 5)

(filter (lambda (n) (> n 3))
        '(1 2 3 4 5))          ;; ===> '(4 5)


(sort '(3 1 2 4 5) <)          ;; ===> '(1 2 3 4 5)

(sort '(3 1 2 4 5) >)          ;; ===> '(5 4 3 2 1)
```

NAME:

"QUIZ 1" (DON'T USE EXPLICIT RECURSION)

```
; Given a list of lists L, combine them
;   all into a single list.
;
; (smush  '((t h i) (s i s) (s o c o) (o l) )
;     ===> '( t h i s i s s o c o o l )
(define (smush L)
```

```
; Add k to each element of L
; (all will be numeric)
;
; (addk 60 '(-18 101 7940))
;        ===> '(42 161 8000)
(define (addk k L)
```

```
; Given two lists, return the number of elements
;   they have in common.
; Hint: use filter and member
(define (matches T W)
```

Careful!

Only one of these works. Why?

```
(define (addk k L)
   (map  (lambda (x) (+ k x))  L))
```

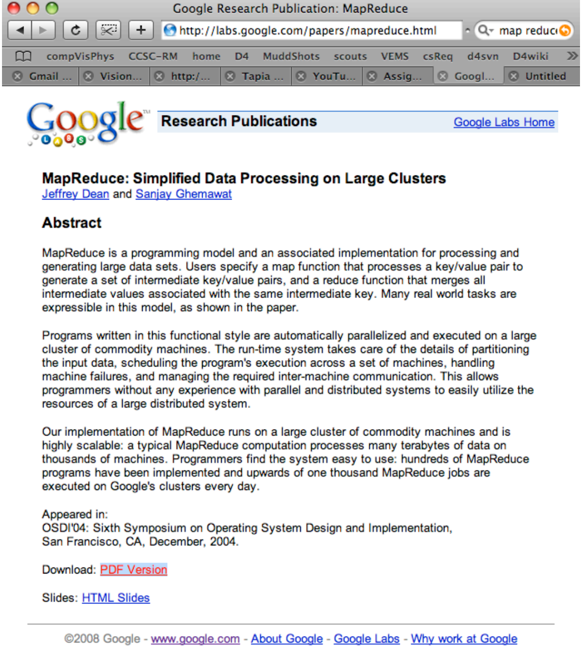
```
(define (addk k L)
   (map  (+ k x)  L))
```

```
(define (addk k L)
   (map  (+ k)  L))
```

MapReduce:
Google's solution
to large-scale
parallel processing:

map: processes
(in parallel)

reduce: combines



Research Publications

Google Labs Home

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

Appeared in:
OSDI'04: Sixth Symposium on Operating System Design and Implementation,
San Francisco, CA, December, 2004.

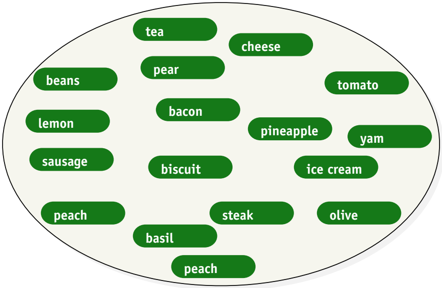
Download: PDF Version

Slides: HTML Slides

©2008 Google - www.google.com - About Google - Google Labs - Why work at Google

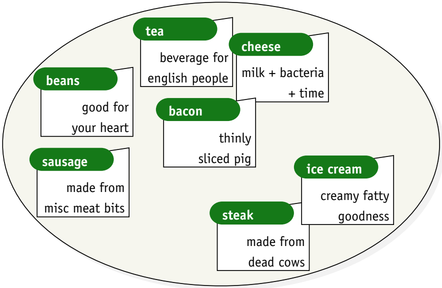
Two Common Interfaces

Set: An unordered collection



What operations do we expect?

Map: Associates "keys" with "values"



What operations do we expect?

Implementation 1: Lists

Sets as unordered lists

```
("tea" "cheese" "tomato" "biscuit" ...)
```

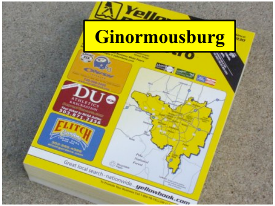
Maps as association lists

```
( ["tea" "beverage for english people]
  ["bacon" "thinly sliced pig"]
  ... )
```

Given this representation, you **already** can write code for set-lookup, map-lookup, set-insert, map-insert, set-delete, map-delete, etc.

But what is the big-O running time for a set/map of size n?

Searching Through a Phone Book

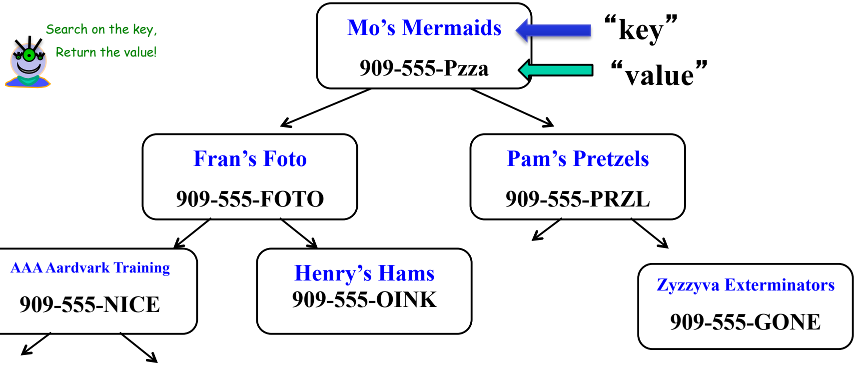


2^{30} entries

```
((“AAA Aardvark Training” “909-555-NICE”)
 (“AAA Alligators” “909-555-BITE”)
  ...
 (“Zyzzyva Exterminators” 909-555-GONE”))
```

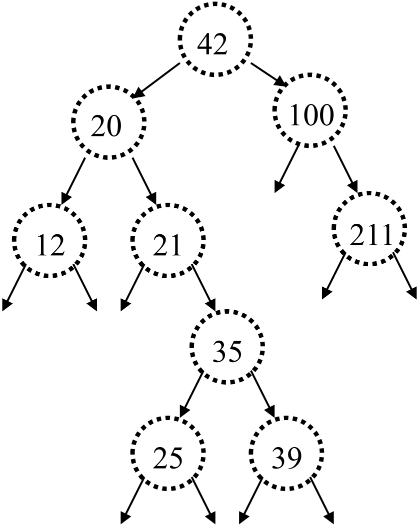
Implementation 2: Binary Search Trees

- ✓ Nodes have *up to* two children
- ✓ Left subtrees have smaller keys; right subtrees have bigger keys.



(Sets are similar, with no "value" in the node)

Binary Search Trees



Identifying Features:

- ✓ Every node has two subtrees
- ✓ Each node has a "key"
- ✓ The root key is always greater than all nodes in a left subtree
- ✓ The root key is always less than all nodes in a right subtree

But Racket only has lists...?