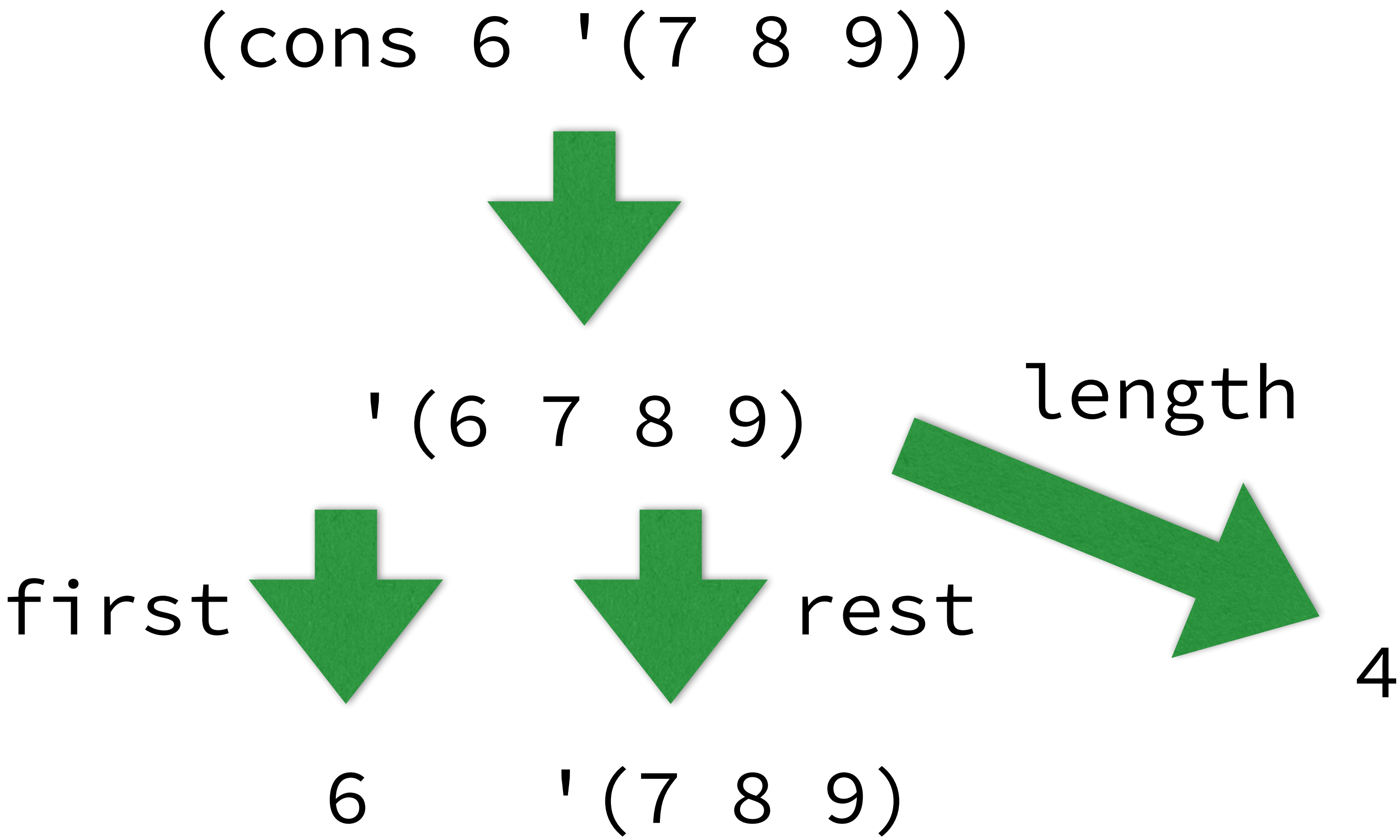
Patterns in Functional Programming

Please take a handout &
Sit in row H or forward

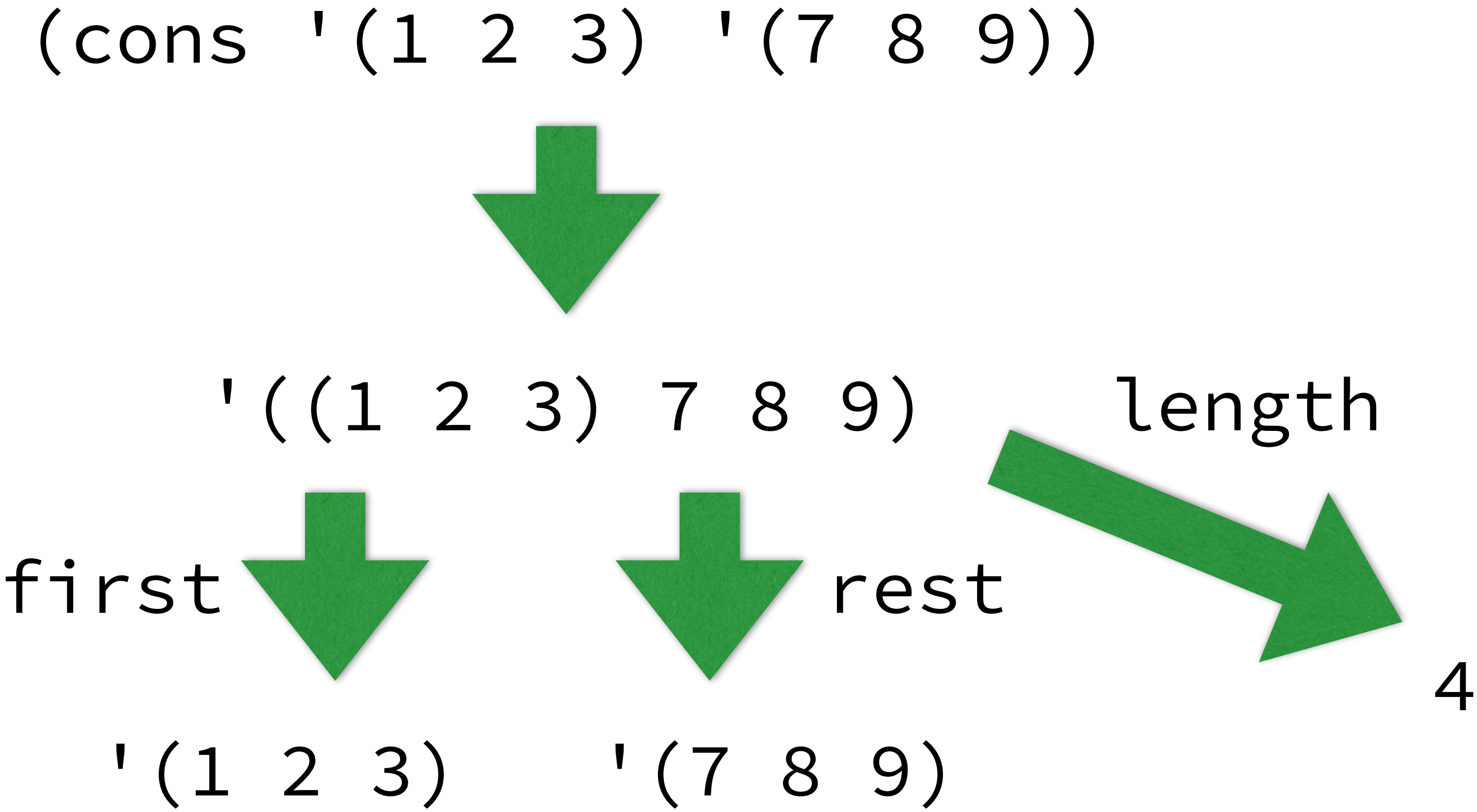
Recap:

Constructing and Deconstructing Lists

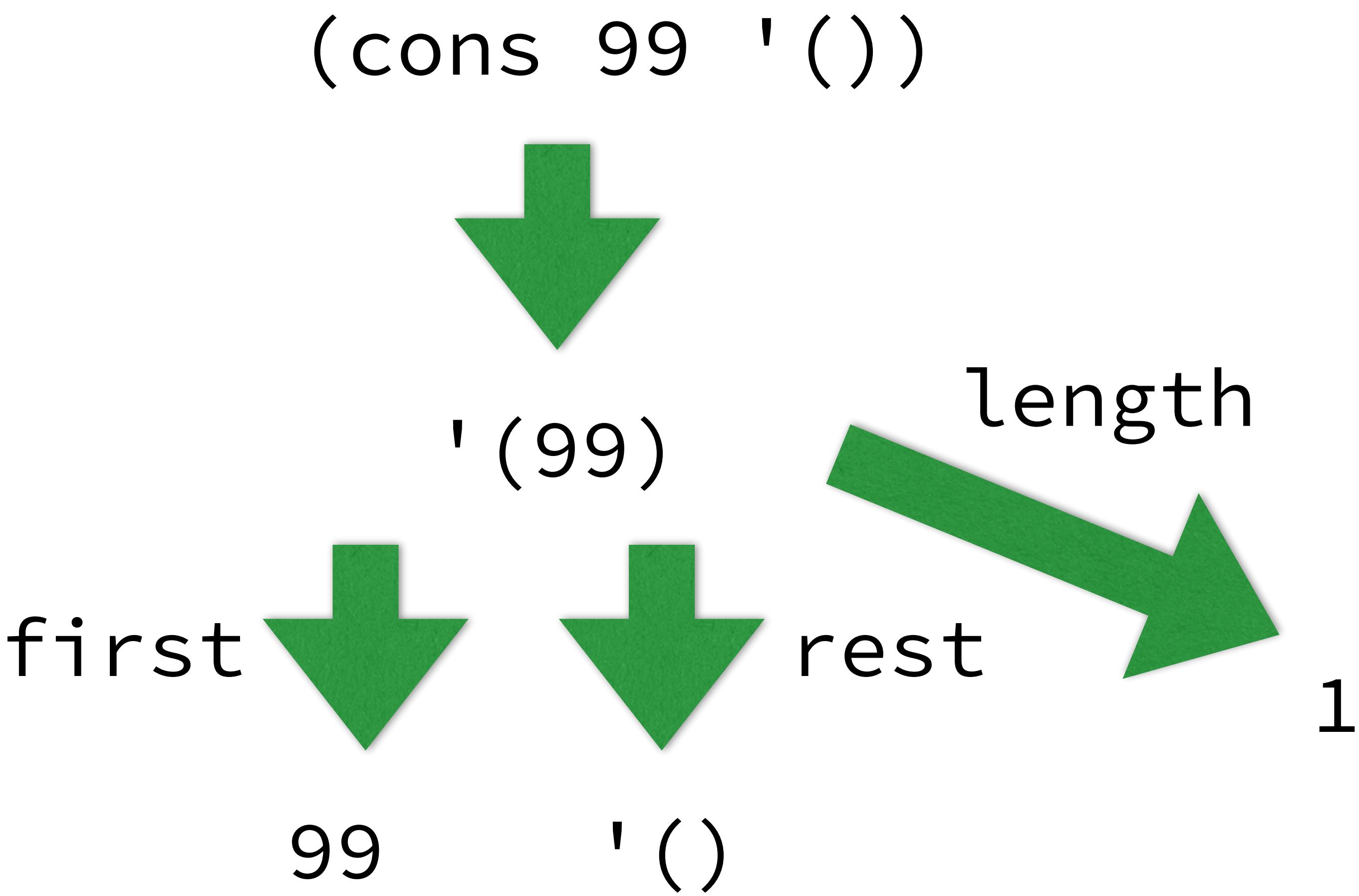
Lists



Lists



Lists



Complexity and Recurrence Relations

In the World of Big O

Asymptotic Analysis

We’re always answering the same question:

How does the algorithm ***scale***
(when we try larger and larger inputs)

NOT

- Exactly how many steps will it execute?
- How many seconds will it take?
- How many megabytes of memory will it need?

Scaling in Time and Space

"Given an array of length n,
the MergeSort algorithm
requires time O(n log n)
and uses O(n) space."

In the limit (for VERY LARGE inputs)

The running time is bounded regardless of the input size.	$O(1)$
An input twice as big takes no more than twice as long.	$O(n)$
An input twice as big takes no more than four times as long.	$O(n^2)$
An input <u>one bigger</u> takes no more than twice as long.	$O(2^n)$

If We Only Care About Scalability…

What are the consequences?

Constant factors can be ignored.

n and **6n** and **200n** scale identically (“linearly”)

Small summands can be ignored.

$\mathbf{n^2}$ and $\mathbf{n^2 + n} + \mathbf{999999}$ are indistinguishable when n is huge.

Grouping Algorithms by Scalability

$O(1)$

takes 6 steps
takes 1 (big) step
no more than 4000 steps
somewhere between 2 and 47 steps, depending on the input

$O(n)$

takes $100n + 3$ steps
takes $n/20 + 10{,}000{,}000$ steps
anywhere between 3 and 68 steps per item, for n items.

$O(n^2)$

takes $2n^2 + 100n + 3$ steps
takes $n^2/17$ steps
somewhere between 1 and 40 steps per item, for n^2 items
anywhere between 1 and 7n steps per item, for n items.

Big O — Step 2

Measuring inputs

How does the algorithm ***scale***
(when we try larger and larger inputs)

The first step is defining a “measure” of input size.
Conventionally we call this n or N, but not always.

Suppose we are analyzing a function
whose input is a single string?

The function is $O(\ldots)$,
where n is the length of the input string.

Big O — Step 2

Measuring inputs

Suppose we are analyzing functions whose input is an array of strings?

The function takes *O(…) time*,
where n is the length of the array.

The function takes O(…) time,
where n is the length of the longest string.

The function takes O(…) time,
where n is the total number of characters.

The function takes O(…) time,
where n is the length of the array
and m is the length of the longest string.

The scalability of `sum` revisited

Step 1: Choose an appropriate measure of input "size"

```
;; input: a list of numbers
;; output: the sum of the given numbers
(define (sum L)
   (if (empty? L)
       0
       (+ (first L) (sum (rest L)))))
```

We care about what happens when the input is large.
For **this** function, how should we measure L?

- The sum of the numbers in L?
- The biggest number in L?
- The total number of digits in the numbers in L?
- Something else?

The scalability of `sum` revisited

Step 2: Choosing what to count

```
;; input: a list of numbers
;; output: the sum of the given numbers
(define (sum L)
   (if (empty? L)
       0
       (+ (first L) (sum (rest L)))))
```

Suppose we're interested in how **running-time** scales.
What should we count?

- Additions ?
- **`empty?`** calls ?
- **`first`** and **`rest`** operations ?
- Maybe all of these ?

Counting Steps

For our asymptotic analysis

```
;; input: a list of numbers
;; output: the sum of the given numbers
(define (sum L)
   (if (empty? L)
       0
       (+ (first L) (sum (rest L)))))
```

Let n be the length of the given list.
We want T(n), the number of "steps" required.

When $n = 0$, we do $\underline{O(1)}$ steps.

When $n > 0$, we do $\underline{O(1)}$ steps
in addition to making a recursive call
on an input of size $\underline{n-1}$.

Recurrence Relations in Big-O

We want to describe a function T(n) such that sum takes O(T(n)) steps on an input of size n

$T(0) = 1$
$T(n) = 1 + T(n-1)$ when $n>0$

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + T(n-1) & \text{otherwise} \end{cases}$$

```
;; Input: n, the length of the input list
;; Output: T(n), the asymptotic number of steps
;;            taken by sum on an input of size n.
(define (T n)
  (if (= n 0)
      1
      (+ 1 (T (- n 1)))))
```

“Solving” the Recurrence

sum requires O(T(n)) steps on an input of size n

```
Welcome to DrRacket, version 6.3 [3m].
Language: racket; memory limit: 128 MB.
> (T 0)
1
> (T 1)
2
> (T 2)
3
> (T 3)
4
> (T 1000)
1001
```

aha! T(n) = n+1 for all inputs
and I (or Mathematica) could prove it!

∴ sum requires O(n) steps for a list of length n

An Error That TDD Might Not Detect

What Changed?

```
;; input: a list of numbers
;; output: the sum of the given numbers
(define (sum L)
   (if (equal? (length L) 0)
       0
       (+ (first L) (sum (rest L)))))
```

Let n be the length of the given list.
We want T(n), the number of “steps” required.

When n = 0, we do <u>O(1)</u> steps.

When n > 0, we do <u>O(n)</u> steps
in addition to making a recursive call
on an input of size <u>n-1</u>.

Recurrence Relation for `sum`

We want to describe a function T(n) such that sum takes O(T(n)) steps on an input of size n

T(0) = 1

T(n) = n + T(n-1)   when n>0

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ n + T(n-1) & \text{otherwise} \end{cases}$$

```
;; Input: n, the length of the input list
;; Output: T(n), the asymptotic number of steps
;;             taken by summ on an input of size n.
(define (T n)
  (if (= n 0)
      1
      (+ n (T (- n 1)))))
```

Solving the New Recurrence for `sum`

```
Welcome to DrRacket, version 6.3 [3m].
Language: racket; memory limit: 128 MB.
> (T 0)
1
> (T 1)
2
> (T 2)
4
> (T 3)
7
> (T 1000)
500501
```

aha! $T(n) = ½n^2+½n+1$ for all inputs
and I (or Mathematica) could prove it!

∴ <u>This</u> `sum` requires $O(n^2)$ steps on an input of size n

Appending Two Lists: Intuition

```
(append '(1 2 3) '(4 5))

   (cons 1 (append '(2 3) '(4 5)))

           (cons 2 (append '(3) '(4 5)))

                   (cons 3 (append '() '(4 5)))

                                   '(4 5)
```

```
(define (append L M)
   (if (empty? L)
       M
       (cons (first L)
             (append (rest L) M))))
```

Appending Two Lists: Intuition

```
(append '(1 2 3) '(4 5)) ===

(cons 1 (append '(2 3) '(4 5))) ===

(cons 1 (cons 2 (append '(3) '(4 5)))) ===

(cons 1 (cons 2 (cons 3 (append '() '(4 5))))) ===

(cons 1 (cons 2 (cons 3 '(4 5)))) ===

'(1 2 3 4 5)
```

Counting Steps

For our asymptotic analysis

```
(define (append L M)
   (if (empty? L)
       M
       (cons (first L)
             (append (rest L) M))))
```

Let n be __the length of list L___.
We want T(n), the number of “steps” required.

When n = 0, we do _O(1)_ steps.

When n > 0, we do _O(1)_ steps
in addition to making a recursive call
on an input of size _n-1_.

The Recurrence Relation

We want to describe a function T(n) such that append takes O(T(n)) steps on an input of size n

T(0) = 1
T(n) = 1 + T(n-1)   when n>0

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + T(n-1) & \text{otherwise} \end{cases}$$

Wait, we've solved this recurrence already!

$\therefore$ append requires O(n) steps
when the **first** input list has length n

upload.wikimedia.org/wikipedia/commons/8/8c/Key_break.jpg

Patterns of Control Flow

double-list

```
(define (double n)
  (* 2 n))

(define (double-list L)
  (if (empty? L)
      '()
      (cons (double (first L))
            (double-list (rest L)))))


; tests
(check-equal? (double-list '()) '())
(check-equal? (double-list '(1)) '(2))
(check-equal? (double-list '(1 2)) '(2 4))
(check-equal? (double-list '(2 3 2)) '(4 6 4))
```

Tracing double-list

```
> (double-list '(1 2 3 4 5))
>(double-list '(1 2 3 4 5))
> (double-list '(2 3 4 5))
> >(double-list '(3 4 5))
> > (double-list '(4 5))
> > >(double-list '(5))
> > > (double-list '())
< < < '()
< < <'(10)
< < '(8 10)
< <'(6 8 10)
< '(4 6 8 10)
<'(2 4 6 8 10)
'(2 4 6 8 10)
```

Tracing `(double-list '(1 2 3 4 5))`

```
> (double-list '(1 2 3 4 5))
>(double-list '(1 2 3 4 5))
> (double-list '(2 3 4 5))
```

```
< '(4 6 8 10)
<'(2 4 6 8 10)
'(2 4 6 8 10)
```

```
(cons (double (first L))
      (double-list (rest L)))
```

square-list

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

(define (square-list L)
  (if (empty? L)
      '()
      (cons (square (first L))
            (square-list (rest L)))))


; tests
(check-equal? (square-list '()) '())
(check-equal? (square-list '(1)) '(1))
(check-equal? (square-list '(1 2)) '(1 4))
(check-equal? (square-list '(2 3 2)) '(4 9 4))
```

The map pattern

```
(define (double-list L)
  (if (empty? L)
      '()
      (cons (double (first L))
            (double-list (rest L)))))

(define (square-list L)
  (if (empty? L)
      '()
      (cons (square (first L))
            (square-list (rest L)))))
```

The map pattern

```
(define (map f L)
  (if (empty? L)
      '()
      (cons (f (first L))
            (map f (rest L)))))
```

```
(define (double-list L)
    (map double L))
```

```
(define (square-list L)
    (map square L))
```

Fun Fact

The "built-in" map function is even more general.

```
> (map double '(1 2 3))
'(2 4 6)
> (map list '(1 2 3) '(10 11 12))
'((1 10) (2 11) (3 12))
> (map + '(1 2 3) '(10 11 12))
'(11 13 15)
```

If we provide two lists,
we also need a two-argument function

sum

```
(define (sum L)
   (if (empty? L)
       0
       (+ (first L) (sum (rest L)))))
```

and what if we wanted product?

```
; tests
(check-equal? (sum '()) 0)
(check-equal? (sum '(1)) 1)
(check-equal? (sum '(1 2)) 3)
(check-equal? (sum '(2 3 2)) 7)
```

product

```
(define (product L)
   (if (empty? L)
       1
       (* (first L) (product (rest L)))))
```

```
; tests
(check-equal? (product '()) 1)
(check-equal? (product '(1)) 1)
(check-equal? (product '(1 2)) 2)
(check-equal? (product '(2 3 2)) 12)
```

The `foldr` pattern

```
(define (foldr f b L)
  (if (empty? L)
      b
      (f (first L) (foldr f b (rest L)))))
```

```
(define (sum L)
    (foldr + 0 L))
```

```
(define (product L)
    (foldr * 1 L))
```

Summing using foldr

```
(sum '(1 2 3))

(foldr + 0 '(1 2 3))

(+ 1 (foldr + 0 '(2 3)))

(+ 2 (foldr + 0 '(3)))

(+ 3 (foldr + 0 '()))

0
```

Summing using foldr

```
(sum '(1 2 3)) ===

(foldr + 0 '(1 2 3)) ===

(+ 1 (foldr + 0 '(2 3))) ===

(+ 1 (+ 2 (foldr + 0 '(3)))) ===

(+ 1 (+ 2 (+ 3 (foldr + 0 '())))) ===

(+ 1 (+ 2 (+ 3 0))) ===

6
```

Define append using foldr

Q

```
(define (append L M)
   (if (empty? L)
       M
       (cons (first L)
             (append (rest L) M))))
```

```
(append '(1 2 3) '(4 5)) ===
   (cons 1 (cons 2 (cons 3 '(4 5))))
```

```
(define (foldr f b L)
  (if (empty? L)
      b
      (f (first L) (foldr f b (rest L)))))
```

```
(define (append L M)
   (foldr cons M L))
```