Checking in

Roughly how much time are you spending on your CS 42 assignment each week?

Firstname Lastname

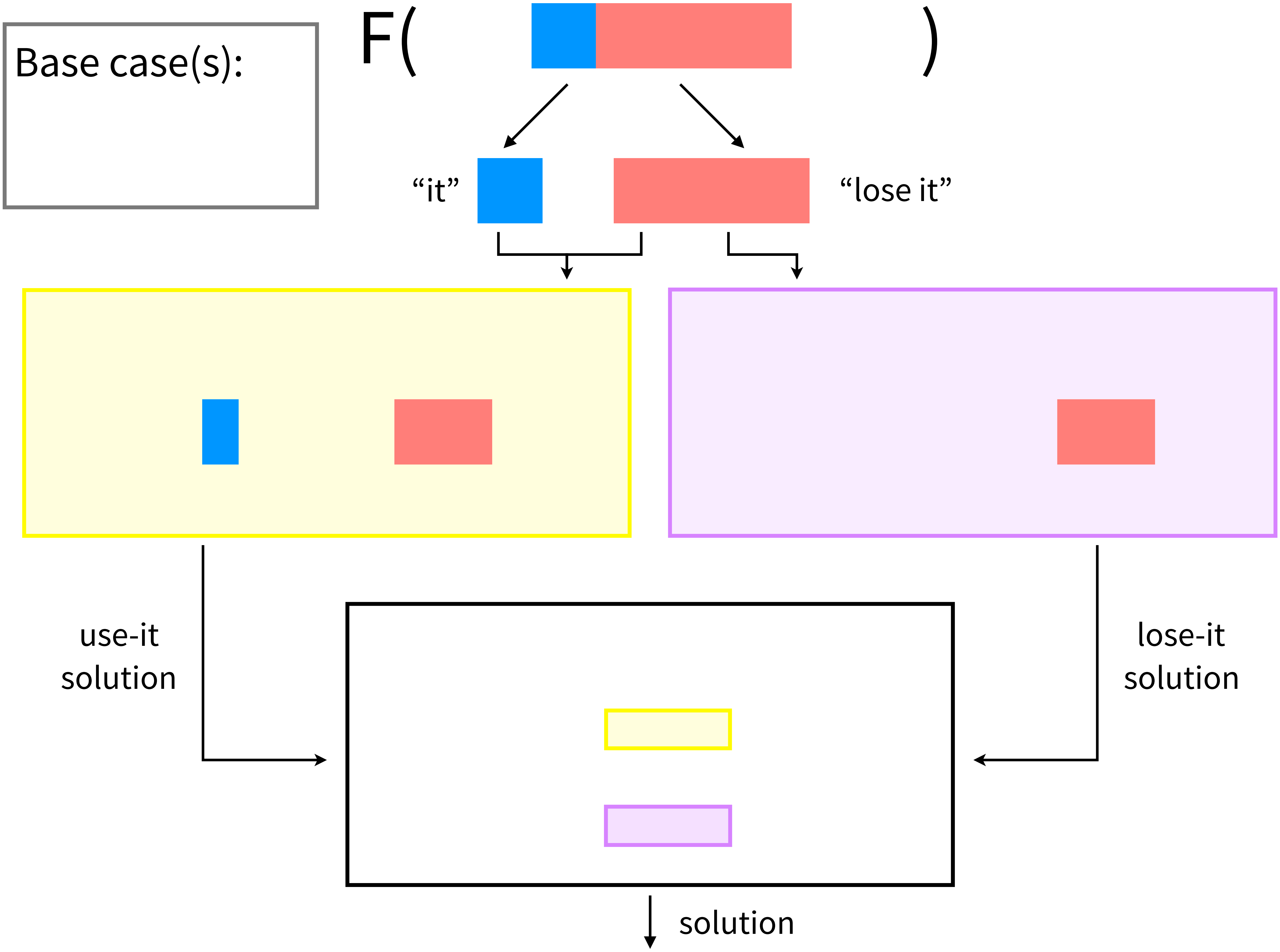
T. 10 / 16

(Your response)

remove

Base case(s):

```
(define (remove e L)
  (if (empty? L)
      empty
      (let* ([it (first L)]
             [lose-it (rest L)]
             [lose-it-solution (remove e lose-it)]
             [use-it-solution (cons it (remove e lose-it))])
        (if (equal? e it)
            lose-it-solution
            use-it-solution))))
```



Base case(s):

F( )

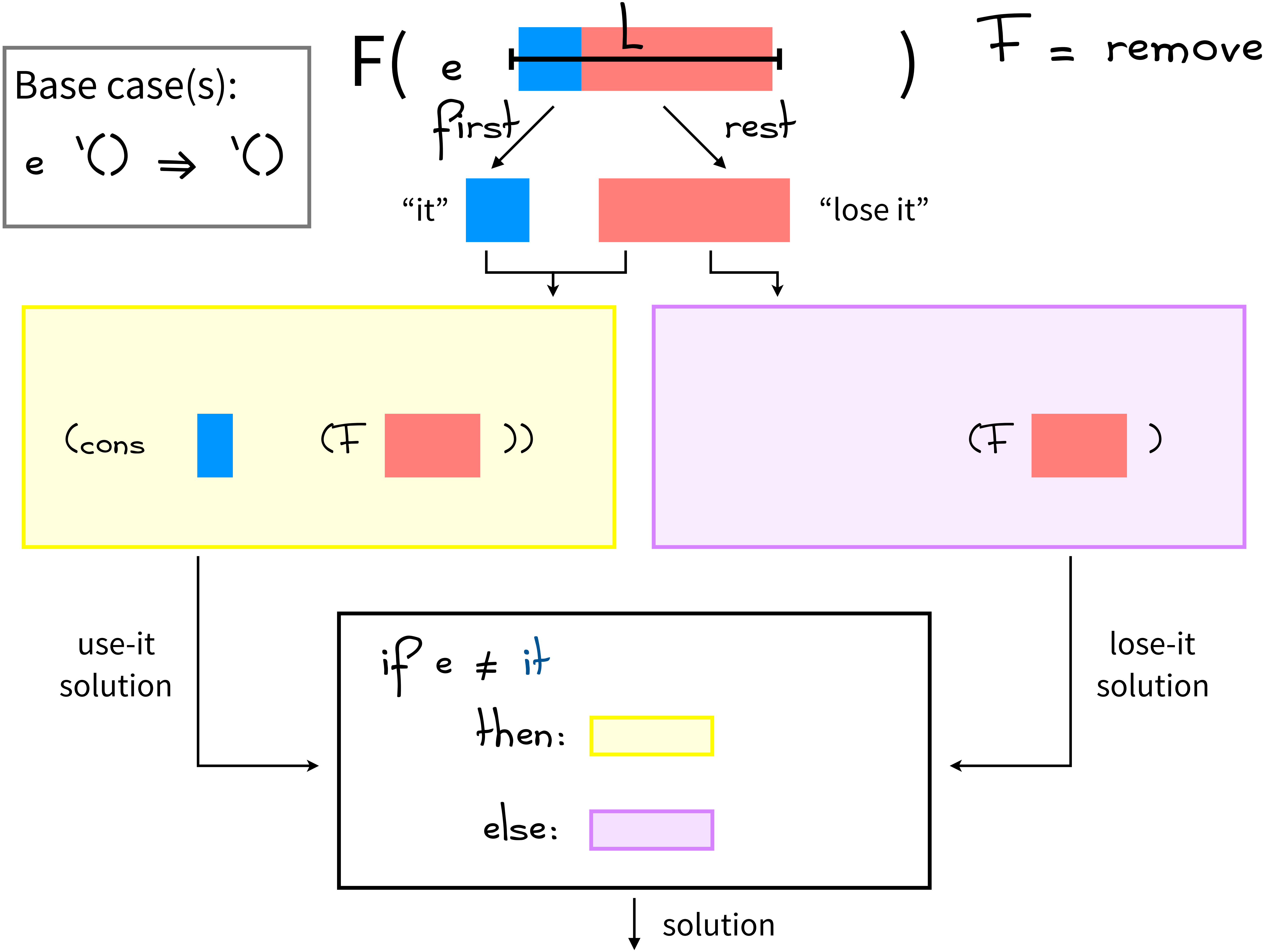
“it”

“lose it”

use-it solution

lose-it solution

solution



Base case(s):

e '() ⇒ '()

F(e L)

F = remove

first

rest

“it”

“lose it”

(cons 🟦 (F 🟥))

(F 🟥)

use-it solution

lose-it solution

if e ≠ it

then: 🟨

else: 🟪

solution

How “good” are these solutions?

Are they efficient?
Do they “cost” more than they should?

Interpreting a theoretical model

Key take-away: it’s **lossy**!

A theory abstracts away certain details.

cost metric:

- corresponds to one “step”
- highlights the essence of the work

  e.g., multiplications, comparisons, function calls…
- serves as a proxy for an empirical measurement

Instead of measuring time, we count steps.

e.g., “This algorithm costs n^2 multiplications.”

Asymptotic Analysis
(Big O)

Asymptotic analysis

We're always answering the same question:

How does the cost ***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?

The informal definition of “Big O”

A *reasonable* upper bound on
(an abstraction of)
a problem’s difficulty or
a solution’s performance,
for *reasonably* large input sizes.

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 one bigger 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.

n^2 and **$n^2 + n$** + **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.

How hard is the problem?

$O(n^n)$
$O(n!)$
$O(2^n)$

Intractable problems
(exponential)

$O(n^3)$
$O(n^2)$
$O(n \log(n))$
$O(n)$

Tractable problems
(polynomial)

$O(\sqrt{n})$
$O(\log(n))$
$O(1)$

No problem!

logs aren’t scary!

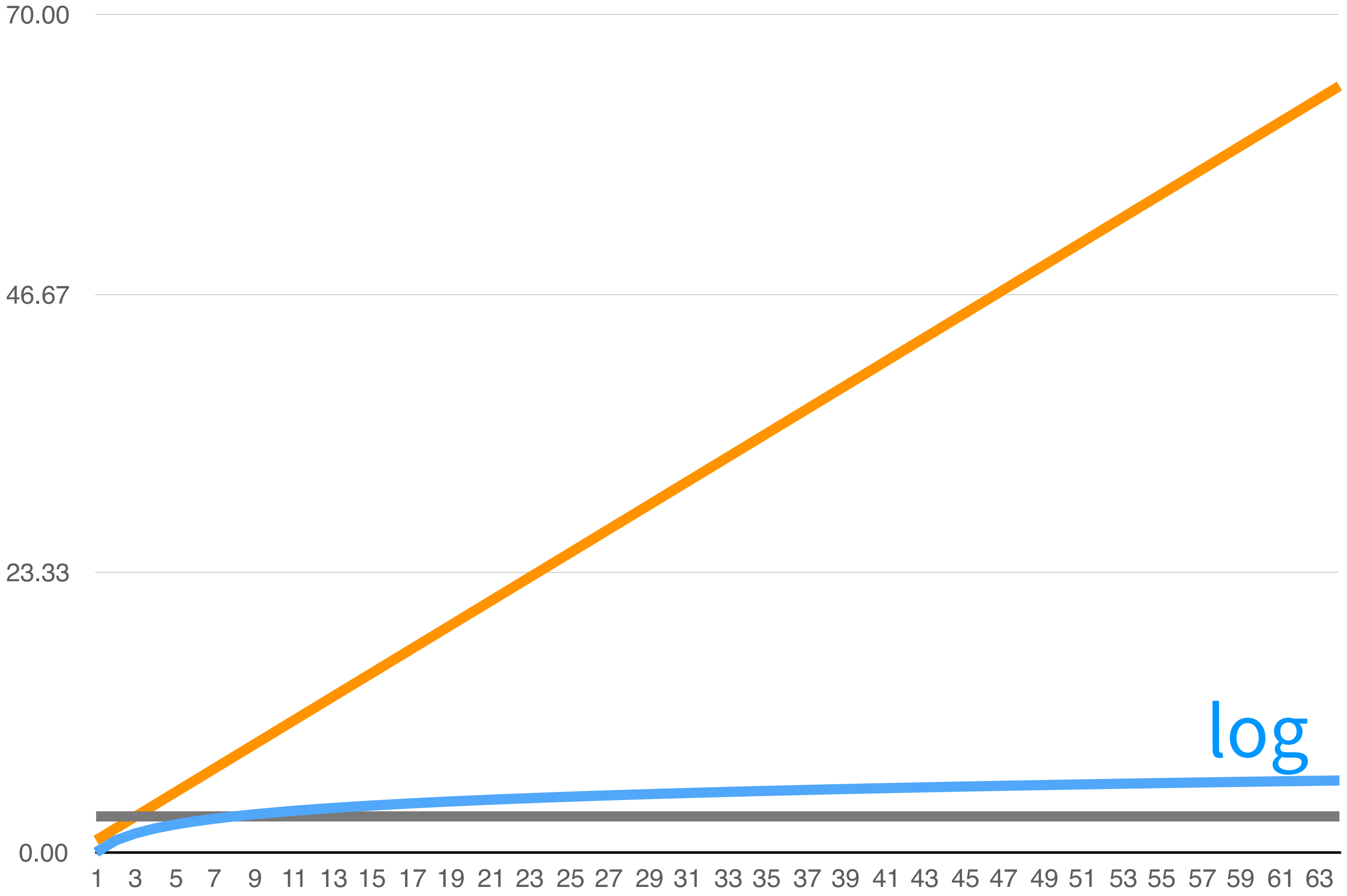
They’re our friends.

$$\log_2 N = p \Leftrightarrow 2^p = N$$

log is the inverse of exponentiation.

How many times can I cut N in half?

Can I avoid looking at *all* the input?!



log2(1) = **0** *// $2^0 = 1$*

log2(2) = **1** *// $2^1 = 2$*

log2(3) ≈ 1.58

log2(4) = **2** *// $2^2 = 4$*

log2(5) ≈ 2.32

log2(6) ≈ 2.58

log2(7) ≈ 2.81

log2(8) = **3** *// $2^3 = 8$*

s-media-cache-ak0.pinimg.com/736x/5d/f7/6d/5df76d1672ccdffc74af2e2bf55330aa.jpg

How hard are these **problems**?

	cost metric	**cost**
`double` multiply a number by 2	multiplications	
`sum` sum a list of numbers	additions	
`half-count` divide a positive number by 2 until you get 1	divisions	

How hard are these **problems**?

	cost metric	**cost**
`double` multiply a number by 2	multiplications	O(1)
`sum` sum a list of numbers	additions	O(n)
`half-count` divide a positive number by 2 until you get 1	divisions	O(log n)

What's the cost, T, for each **solution?**

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

```
(define (sum n)
  (if (= n 0)
      0
      (+ n (sum (- n 1)))))
```

```
(define (half-count n)
  (if (= n 1)
      0
      (+ 1 (half-count (quotient n 2)))))
```

input size

	double multiplications	**sum** additions	**half-count** divisions
$T(0)$			n/a
$T(1)$			
$T(2)$			
$T(3)$			
$T(4)$			
…			
$T(n)$			

What’s the cost, T, for each **solution**?

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

```
(define (sum n)
  (if (= n 0)
      0
      (+ n (sum (- n 1)))))
```

```
(define (half-count n)
  (if (= n 1)
      0
      (+ 1 (half-count (quotient n 2)))))
```

input size

	double multiplications	**sum** additions	**half-count** divisions
$T(0)$	1	0	n/a
$T(1)$	1	1	0
$T(2)$	1	2	1
$T(3)$	1	3	1
$T(4)$	1	4	2
…	…	…	…
$T(n)$	1	n	$\lfloor \log_2 n \rfloor$

Can we prove it?

https://www.simplelongboards.com/boards/platypus/

Recurrence Relations

(translating code to math)

Translating recursion to recurrence relations

For a given cost metric: additions

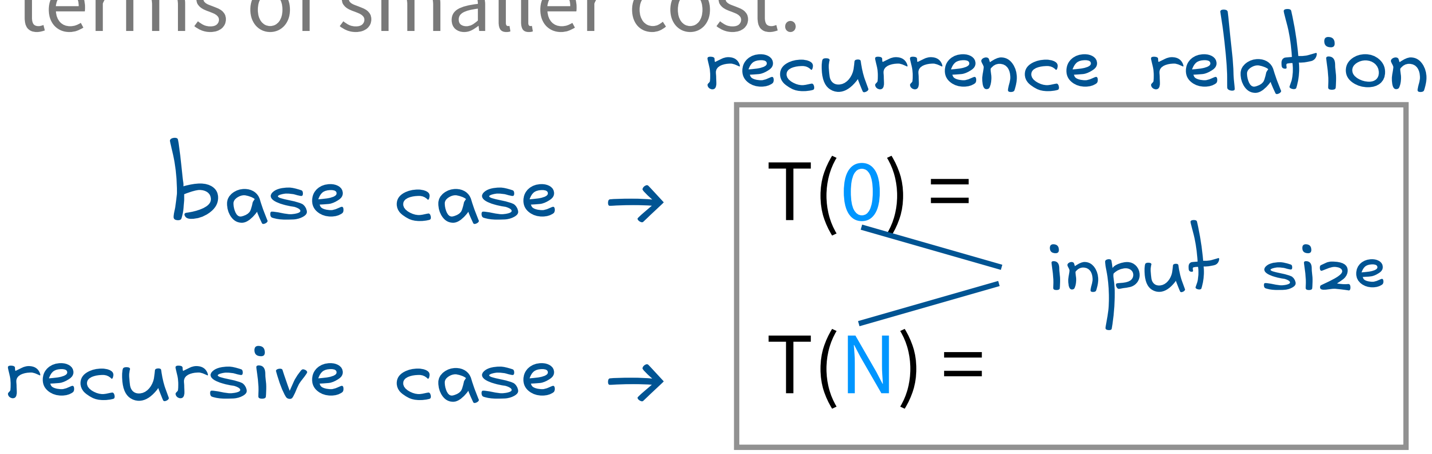
1. Translate the base case(s), using specific input sizes

   How many steps does this base case take?

2. Translate the recursive case(s), using input size N

   Define T(N) recursively, in terms of smaller cost.

```
(define (sum n)
  (if (= n 0)
      0
      (+ n (sum (- n 1)))))
```



T(0) =

T(N) =

Translating recursion to recurrence relations

For a given cost metric: additions

1. Translate the base case(s), using specific input sizes

   How many steps does this base case take?

2. Translate the recursive case(s), using input size N

   Define T(N) recursively, in terms of smaller cost.

```
(define (sum n)
  (if (= n 0)
      0
      (+ n (sum (– n 1)))))
```

recurrence relation

base case → $T(0) = 0$

input size

recursive case → $T(N) = 1 + T(N-1)$

$$
\begin{aligned}
T(N) &= 1 + T(N-1) \\
&= 1 + 1 + T(N-2) \\
&= 1 + 1 + 1 + T(N-3) \\
&\ldots \\
&= 1 + 1 + 1 + \ldots 1 + T(N-N)
\end{aligned}
$$

$$
\begin{aligned}
&= 1*1 + T(N-1) \\
&= 2*1 + T(N-2) \\
&= 3*1 + T(N-3) \\
&\ldots \\
&= N*1 + T(N-N) = N \in O(N)
\end{aligned}
$$

closed form

asymptotic form

Translating recursion to recurrence relations

For a given cost metric: arithmetic operations and comparisons

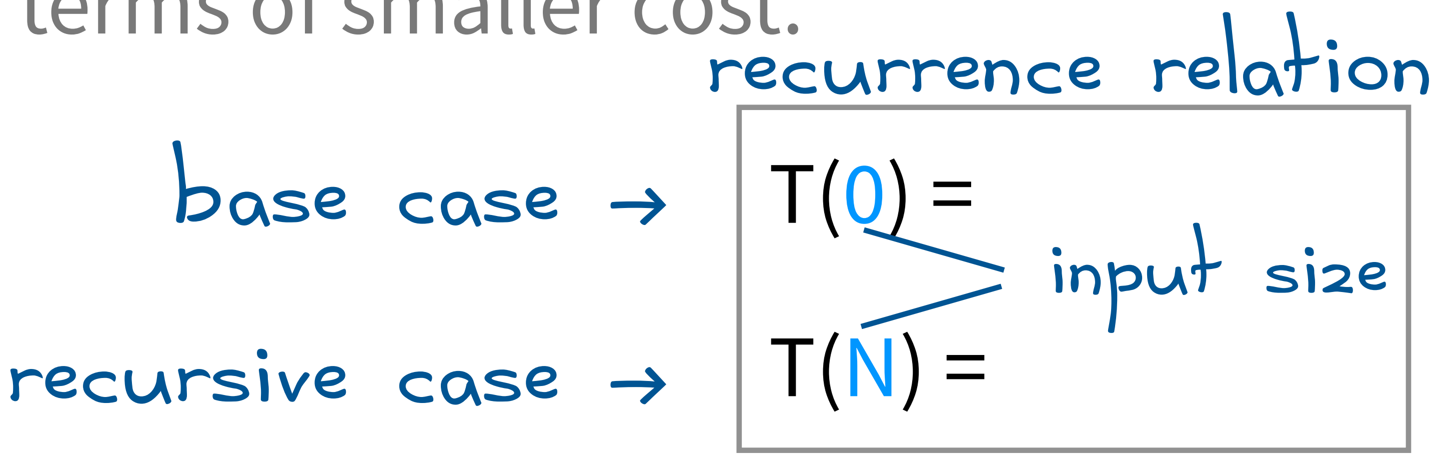
1. Translate the base case(s), using specific input sizes

   How many steps does this base case take?

2. Translate the recursive case(s), using input size N

   Define T(N) recursively, in terms of smaller cost.

```
(define (sum n)
  (if (= n 0)
      0
      (+ n (sum (- n 1)))))
```



T(0) =

T(N) =

Translating recursion to recurrence relations

For a given cost metric: arithmetic operations and comparisons

1. Translate the base case(s), using specific input sizes

   How many steps does this base case take?

2. Translate the recursive case(s), using input size N

   Define T(N) recursively, in terms of smaller cost.

```
(define (sum n)
  (if (= n 0)
      0
      (+ n (sum (- n 1)))))
```

recurrence relation

base case → $T(0) = 1$

input size

recursive case → $T(N) = 3 + T(N-1)$

$$
\begin{aligned}
T(N) &= 3 + T(N-1) \\
&= 3 + 3 + T(N-2) \\
&= 3 + 3 + 3 + T(N-3) \\
&\dots \\
&= 3 + 3 + 3 + \dots 3 + T(N-N)
\end{aligned}
\qquad
\begin{aligned}
&= 1*3 + T(N-1) \\
&= 2*3 + T(N-2) \\
&= 3*3 + T(N-3) \\
&\dots \\
&= N*3 + T(N-N) = 3N + 1 \in O(N)
\end{aligned}
$$

closed form

asymptotic form

Translating recursion to recurrence relations

For a given cost metric: divisions

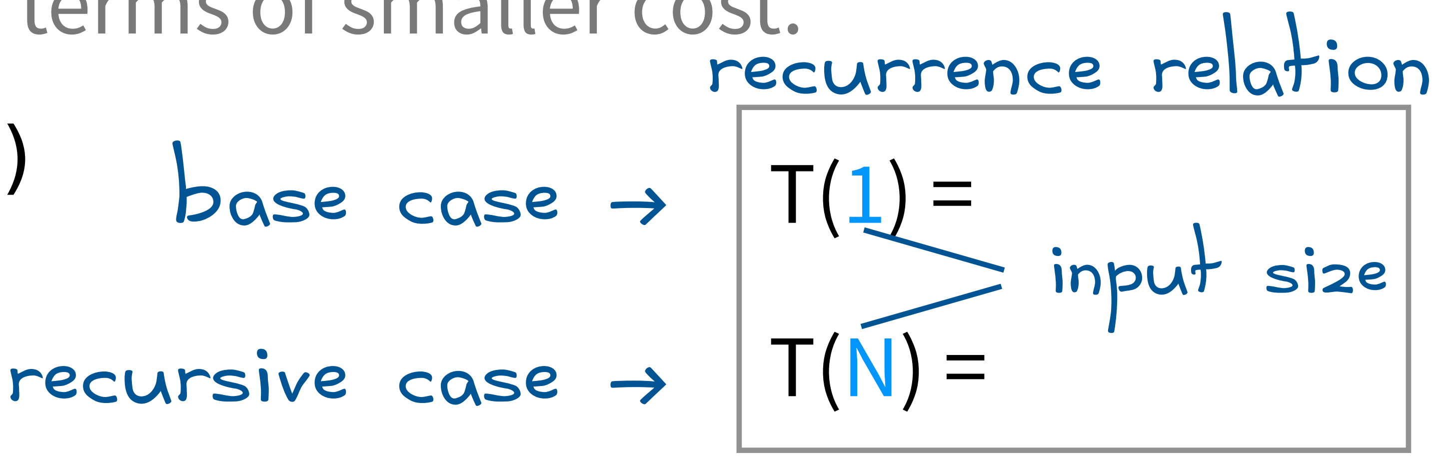
1. Translate the base case(s), using specific input sizes

   How many steps does this base case take?

2. Translate the recursive case(s), using input size N

   Define T(N) recursively, in terms of smaller cost.

```
(define (half-count n)
  (if (= n 1)
      0
      (+ 1 (half-count (quotient n 2)))))
```



recurrence relation

base case →

T(1) =

input size

recursive case →

T(N) =

Translating recursion to recurrence relations

For a given cost metric: divisions

1. Translate the base case(s), using specific input sizes

   How many steps does this base case take?

2. Translate the recursive case(s), using input size N

   Define T(N) recursively, in terms of smaller cost.

```
(define (half-count n)
  (if (= n 1)
      0
      (+ 1 (half-count (quotient n 2)))))
```

recurrence relation

base case → $T(1) = 0$

input size

recursive case → $T(N) = 1 + T(N/2)$

$$
\begin{aligned}
T(N) &= 1 + T(N/2) \\
&= 1 + 1 + T(N/4) \\
&= 1 + 1 + 1 + T(N/8) \\
&\dots \\
&= 1 + 1 + 1 + \dots 1 + T(N/N)
\end{aligned}
$$

$$
\begin{aligned}
&= 1 + T(N/2) \\
&= 2 + T(N/4) \\
&= 3 + T(N/8) \\
&\dots \\
&= \log_2 N + T(N/N) = \log_2 N \in O(\log N)
\end{aligned}
$$

closed form

asymptotic form

How hard are these problems?

	cost metric	**work to do**	**predicted cost**
`remove`	number of comparisons made	visit every element	O(n)
`uniq`	number of comparisons made	compare each element to all the other elements	O(n^2)
`sublists`	number of sublists created	construct 2^n lists	O(2^n)

The cost of `remove`

measured in **list-element comparisons**

```
(define (remove e L)
  (if (empty? L)
      empty
      (let* ([it (first L)]
             [lose-it (rest L)]
             [lose-it-solution (remove e lose-it)]
             [use-it-solution (cons it lose-it-solution)])
        (if (equal? e it)
            lose-it-solution
            use-it-solution))))
```

$$T(0) = 0$$

$$T(N) = \boxed{1} + \boxed{T(N\text{-}1)}$$

$$\begin{aligned} T(N) &= 1 + T(N\text{-}1) \\ &= 1 + 1 + T(N\text{-}2) \\ &\in O(N) \end{aligned}$$

The cost of `uniq`

measured in **list-element comparisons**

```
(define (uniq L)
  (if (empty? L)
      '()
      (let* ([it        (first L)]
             [lose-it (rest L)]
             [lose-it-soln (uniq lose-it)]
             [use-it-soln  (cons it lose-it-soln)])
        (if (member it lose-it-soln)
            lose-it-soln
            use-it-soln))))
```

N-1
comparisons

$$T(0) = 0$$

$$T(N) = \boxed{N\text{-}1} + \boxed{T(N\text{-}1)}$$

$$\begin{aligned} T(N) &= N\text{-}1 + T(N\text{-}1) \\ &= N\text{-}1 + N\text{-}2 + T(N\text{-}2) \\ &\in O(N^2) \end{aligned}$$

The cost of `sublists`

measured in number of sublists created, i.e., calls to **`cons`** and **`empty`**

```
(define (sublists L)
  (if (empty? L)
      (list empty)
      (let* ([it      (first L)]
             [lose-it (rest L)]
             [lose-it-soln (sublists lose-it)]
             [use-it-soln  (map (lambda (l) (cons it l))
                                 lose-it-soln)])
        (append use-it-soln lose-it-soln))))
```

2^{N-1} elements

2^{N-1} cons-es

$$T(0) = \boxed{1}$$

$$T(N) = \boxed{2^{N-1}} + \boxed{2^{N-1}} + \boxed{T(N-1)}$$

$$= 2^N + T(N-1)$$

$$T(N) \in O(2^N)$$

Problems *vs* solutions

	cost metric	**work to do**	**predicted cost**	**UIoLI cost**
`remove`	number of comparisons made	visit every element	$O(n)$	$O(n)$
`uniq`	number of comparisons made	compare each element to all the other elements	$O(n^2)$	$O(n^2)$
`sublists`	number of sublists created	construct 2^n lists	$O(2^n)$	$O(2^n)$