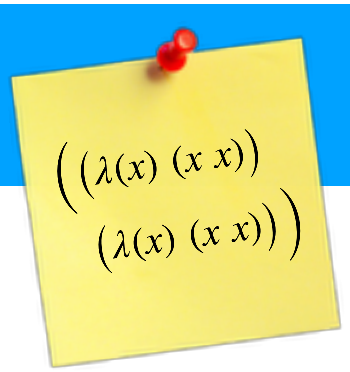
Programming with Recursion and Symbolic Expressions

CIS 352 — Spring 2020

Kris Micinski

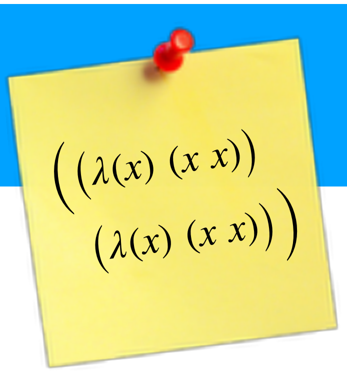
Example



Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

Example

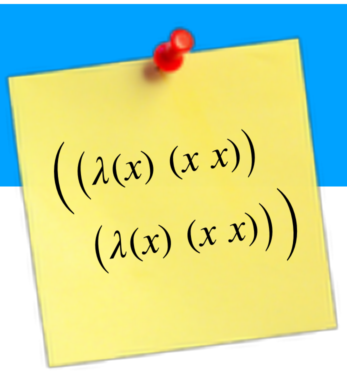


Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

Defines **base case**

Example

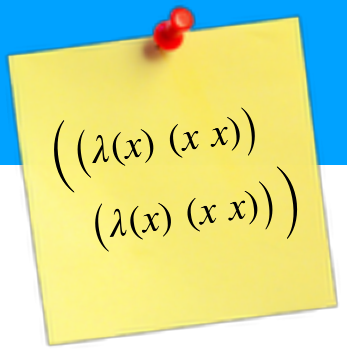


Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

and **inductive / recursive** case

Example

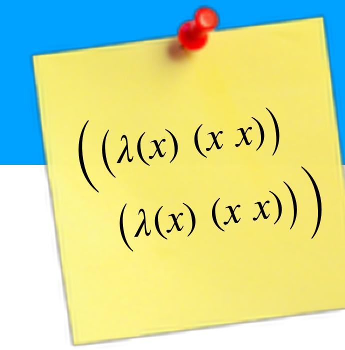



```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as “substitution”

```
> (factorial 2)
```

Example



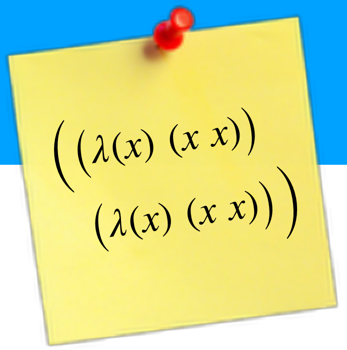

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as “substitution”

```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
```

Copy defn, substitute for argument `n`

Example



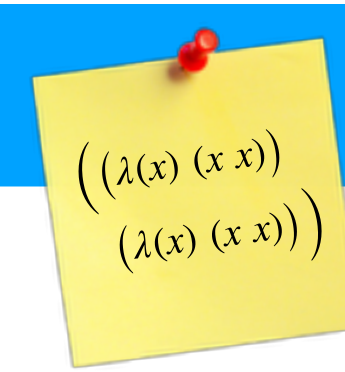

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as "substitution"

```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
= (if #t 1 (* 2 (factorial (sub1 2))))
```

Evaluate `if`

Example

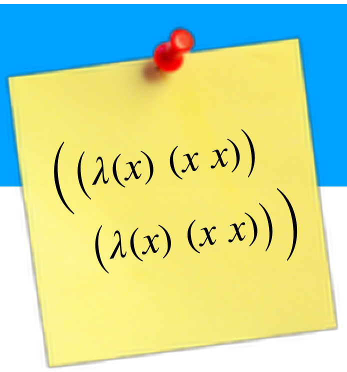



```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as "substitution"

```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
= (if #t 1 (* 2 (factorial (sub1 2))))
= (* 2 (factorial (sub1 2)))
```

Example



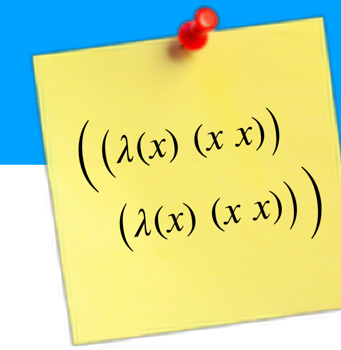

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as "substitution"

```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
= (if #t 1 (* 2 (factorial (sub1 2))))
= (* 2 (factorial (sub1 2)))
= (* 2 (factorial 1))
```

Evaluate `sub1`

Example



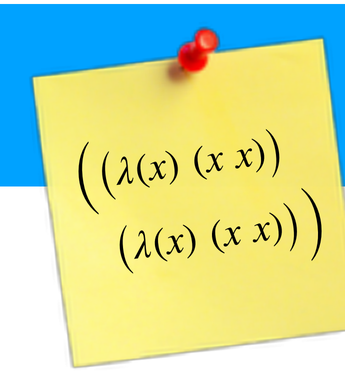

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as "substitution"

```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
= (if #t 1 (* 2 (factorial (sub1 2))))
= (* 2 (factorial (sub1 2)))
= (* 2 (factorial 1))
= (* 2 (if (= 1 0)
        1
        (* n (factorial (sub1 1))))
```

Substitute (again)

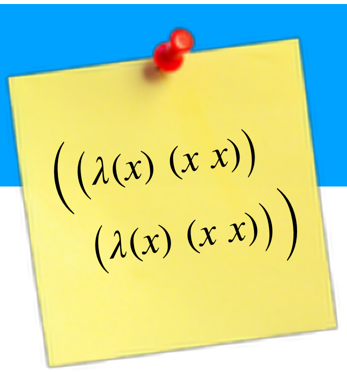
Example




```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))


= (* 2 (if (= 1 0)
           1
           (* 1 (factorial (sub1 1)))))
= (* 2 (* 1 (factorial (sub1 1))))
= (* 2 (* 1 (factorial 0)))
= (* 2 (* 1 (if (= 0 0) 1 …)))
= (* 2 (* 1 (if #t 1 …)))
= (* 2 (* 1 1))
= (* 2 1)
= 2
```

Example



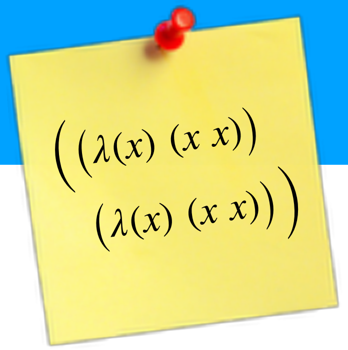

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))

= (* 2 (if (= 1 0)
           1
           (* 1 (factorial (sub1 1)))))
= (* 2 (* 1 (factorial (sub1 1))))
= (* 2 (* 1 (factorial 0)))
= (* 2 (* 1 (if (= 0 0) 1 …)))
= (* 2 (* 1 (if #t 1 …)))
= (* 2 (* 1 1))
= (* 2 1)
= 2
```

This is “textual reduction” semantics

More on this later

Example




```
…
= (* 2 (if (= 2 0)
           1
           (* n (factorial (sub1 2)))))
= (* 2 (factorial 1))
= …
= (* 2 (* 1 1))
= (* 2 1)
= 2
```

Notice we're building a big stack of calls to ∗

Then recursion "bottoms out:" returns back to finish the work

(More on this next week…)

Exercise

Complete the following substitution for `(log2 2)`

```
(define (log2 n)
  (if (= n 1) 0 (+ 1 (log2 (/ n 2)))))

  (log2 2)
= (if (= 2 1) 0 (+ 1 (log2 (/ 2 2))))
= ???
= …
= ???
```

Exercise

```
(define (log2 n)
  (if (= n 1) 0 (+ 1 (log2 (/ n 2)))))
```

```
  (log2 2)
= (if (= 2 1) 0 (+ 1 (log2 (/ 2 2))))
= (+ 1 (log2 (/ 2 2)))
= (+ 1 (log2 1))
= (+ 1 (if (= 1 1) 0 (+ 1 (log2 (/ 1 2)…)
= (+ 1 (if #t 0 (+ 1 (log2 (/ 1 2)…)
= (+ 1 0)
= 1
```

Exercise

Write the definition of `(fib n)` in Racket using the following definition:

$$\text{fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

Exercise

Answer (one of many)

```
(define (fib n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Exercise

Question: what is the big-O time complexity of this implementation?

```
(define (fib n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

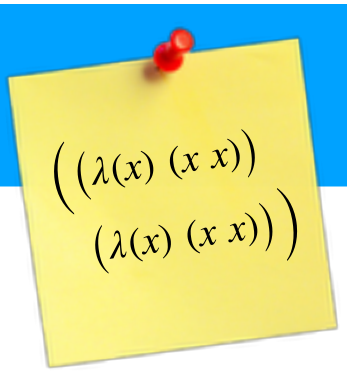
Exercise

Answer: $O(2^n)$ or *exponential*

(Fun fact: actually φ^n, where φ is the golden ratio)

```
(define (fib n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

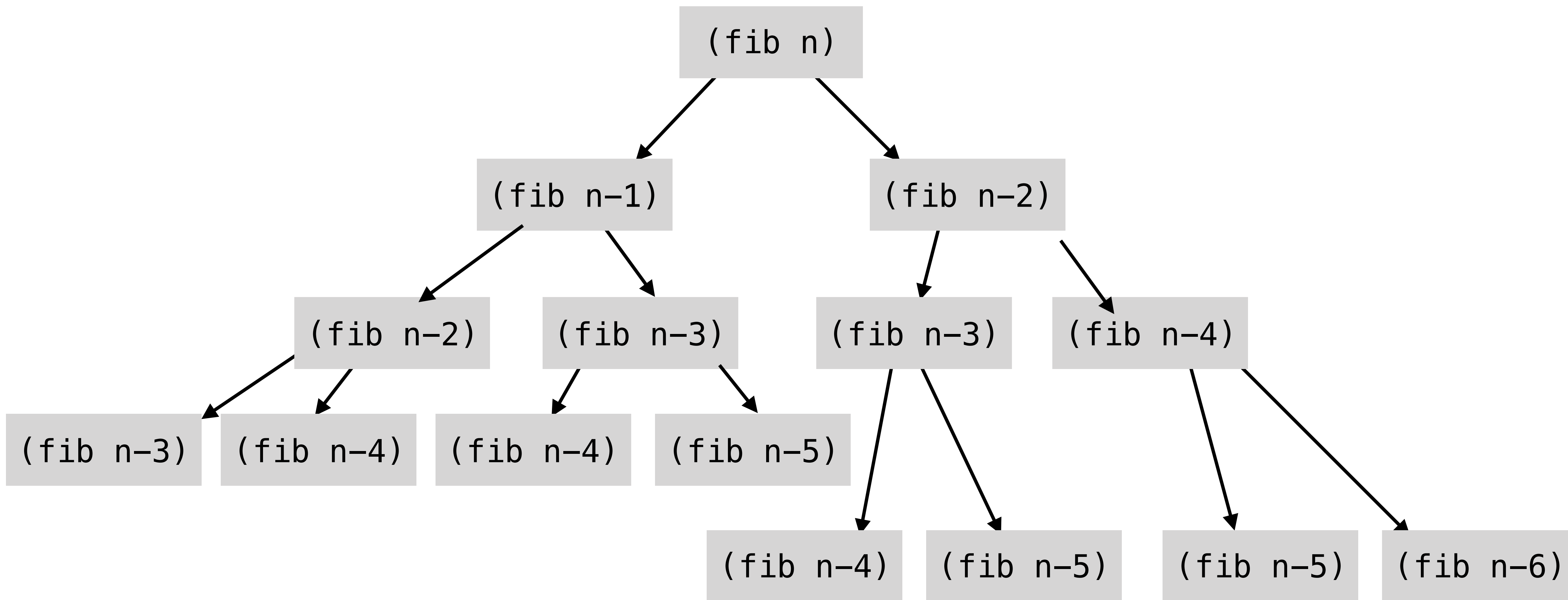
Example

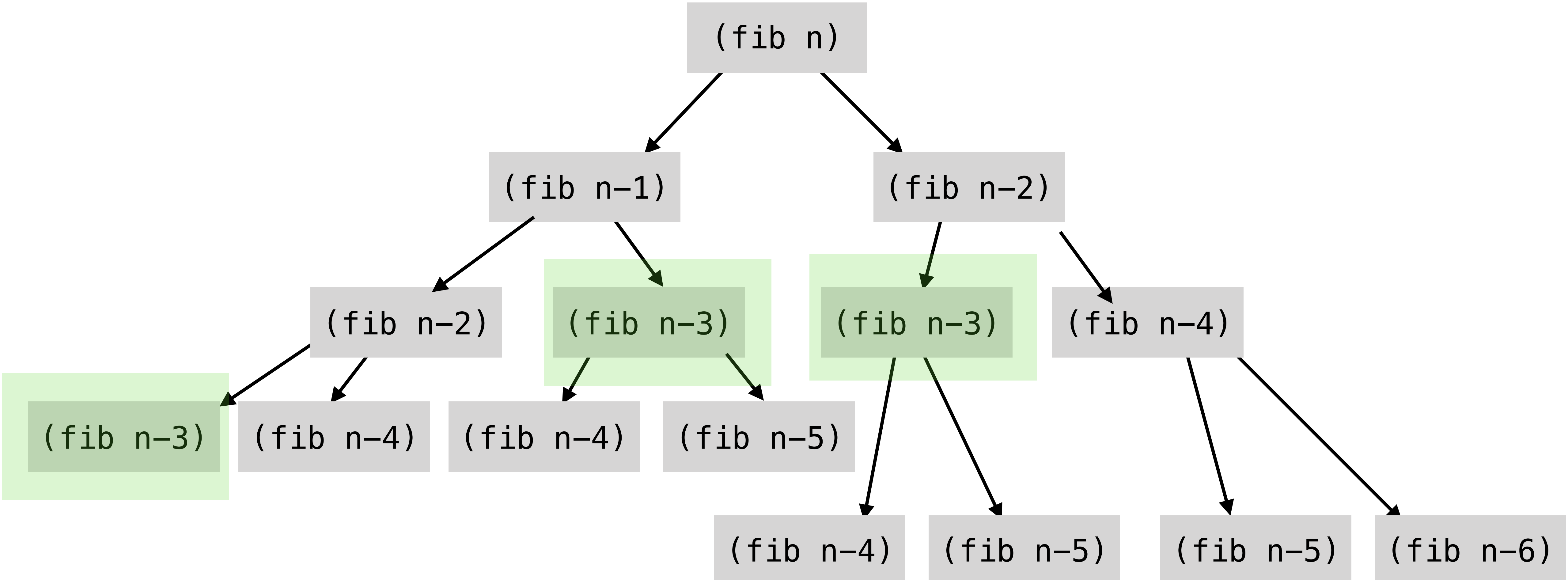


We say that this algorithm uses a "top-down" approach

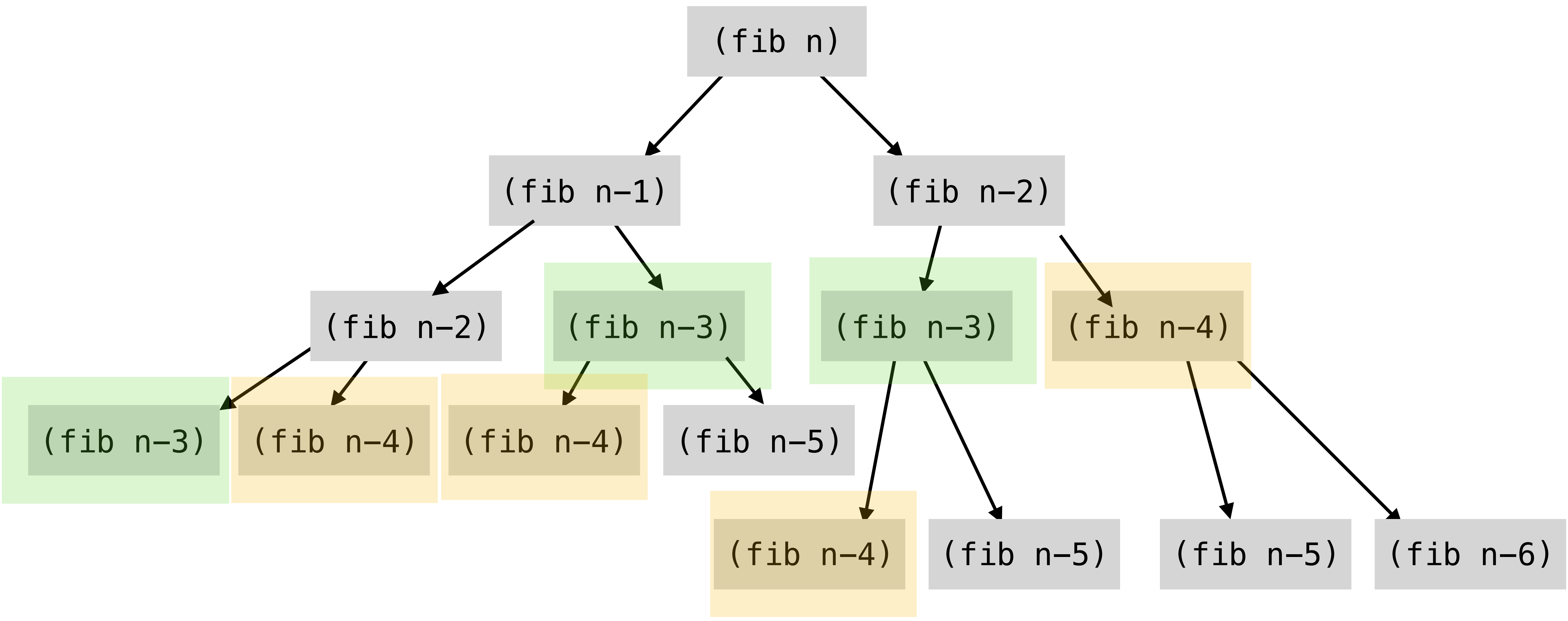
```
(define (fib n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Because it calculates each number by first calculating the previous two fibonacci numbers


(fib n)
(fib n−1)
(fib n−2)
(fib n−2)
(fib n−3)
(fib n−3)
(fib n−4)
(fib n−3)
(fib n−4)
(fib n−4)
(fib n−5)
(fib n−4)
(fib n−5)
(fib n−5)
(fib n−6)

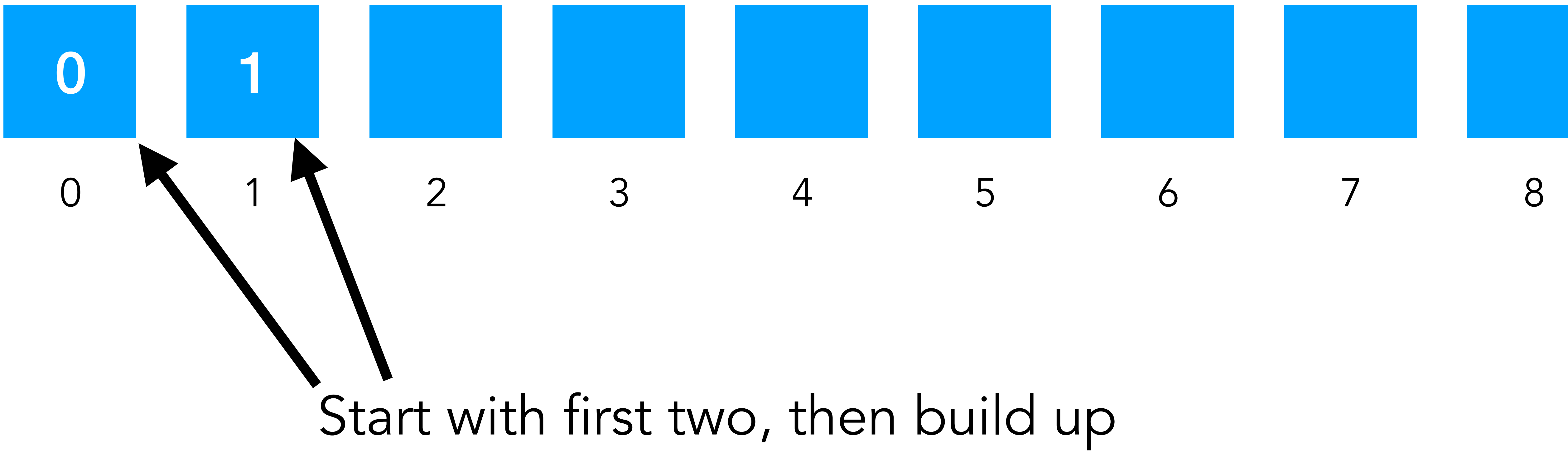

(fib n)
(fib n-1)
(fib n-2)
(fib n-2)
(fib n-3)
(fib n-3)
(fib n-4)
(fib n-3)
(fib n-4)
(fib n-4)
(fib n-5)
(fib n-4)
(fib n-5)
(fib n-5)
(fib n-6)

etc…

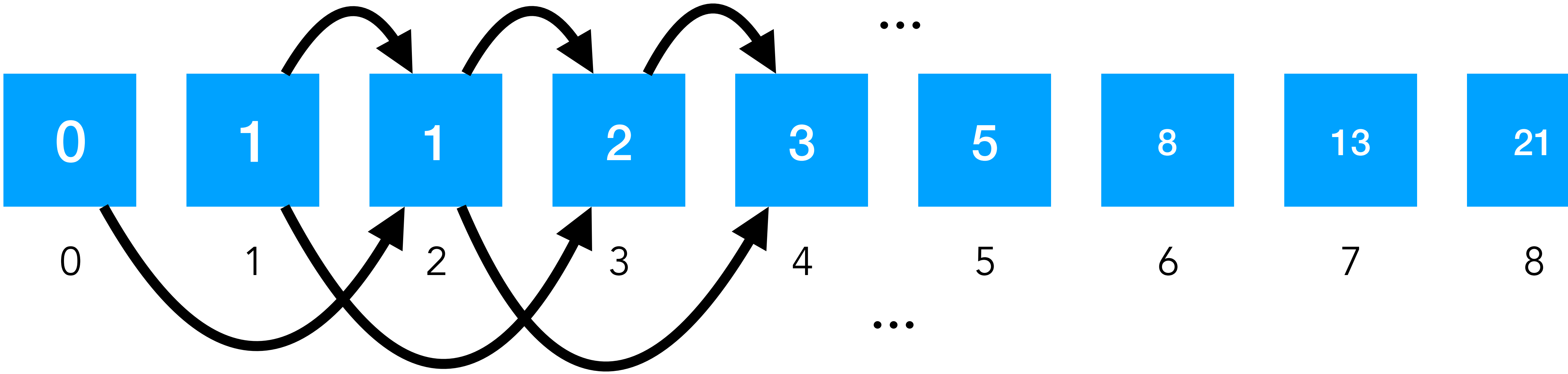


Lots of redundant work

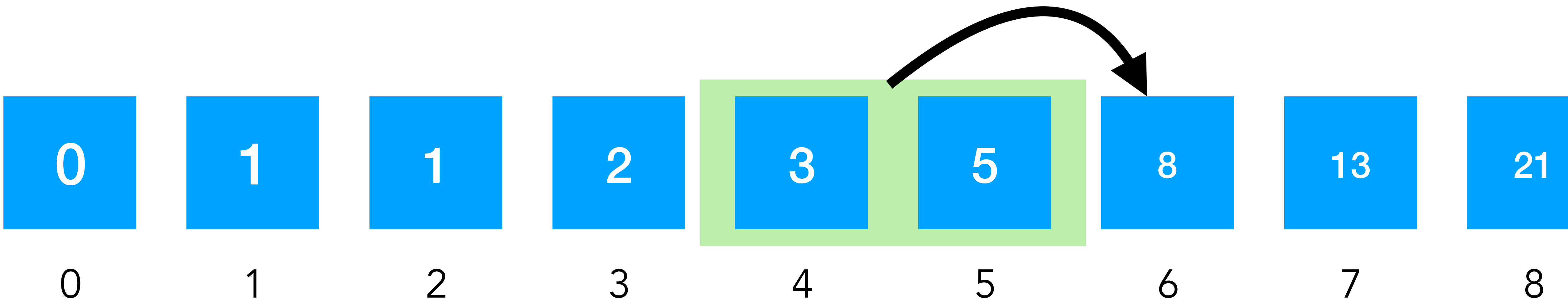
Instead, use ***dynamic programming:***
design a recursive solution top-down, but implement as a bottom-up algorithm!



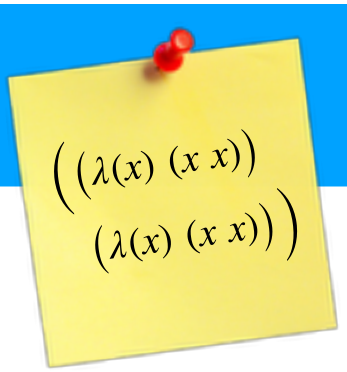
Instead, use ***dynamic programming:***
design a recursive solution top-down, but implement as a bottom-up algorithm!



Key idea: only need to look at **two most recent** numbers



Example



Accumulate via arguments

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Exercise

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Question: what is the runtime complexity of `fib`?

Exercise

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Answer: O(n), fib-helper runs from `n` to `0`

Consider how `fib-h` executes

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

```
(fib-helper 3 0 1)
= (if (= 3 0) 0 (fib-h (- 3 1) 1 (+ 0 1)))
= …
= (fib-h 2 1 1)
= (if (= 2 0) 1 (fib-h (- 2 1) 1 (+ 1 1)))
= …
= (fib-h 1 1 2)
```

Notice that we don't get the "stacking" behavior: recursive calls don't grow the stack

This is because `fib-h` is **tail recursive**

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Intuitively: a callsite is in **tail-position** if it is the **last thing** a function will do before exiting

(We call these **tail calls**)

This is because `fib-h` is **tail recursive**

Both of these are tail calls

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Intuitively: a callsite is in **tail-position** if it is the **last thing** a function will do before exiting

(We call these **tail calls**)

Tail calls / tail recursion

- Unlike calls in general, ***tail calls*** do not affect the stack:
  - Tail calls *do not grow* (or shrink) the stack.
    - They are more like a goto/jump than a normal call.
- A subexpression is in ***tail position*** if it's the last subexpression to run, whose return value is also the value for its parent expression:
  - In `(let ([x rhs]) body)`; <u>body</u> is in *tail position…*
  - In `(if grd thn els)`; <u>thn</u> & <u>els</u> are in *tail position…*
- A function is ***tail recursive*** if all recursive calls in tail position
- Tail-recursive functions are analogous to loops in imperative langs

Exercise

Which of the following is tail recursive?

```
(define (length-0 l)
  (if (null? l)
      0
      (+ 1 (length-0 (cdr l)))))

(define (length-1 l n)
  (if (null? l)
      n
      (length-1 (cdr l) (+ n 1))))
```

Exercise

Answer

```
(define (length-0 l)
  (if (null? l)
      0
      (+ 1 (length-0 (cdr l)))))
```

Not tail recursive

Adds (+ 1 _) operation to stack

```
(define (length-1 l n)
  (if (null? l)
      n
      (length-1 (cdr l) (+ n 1))))
```

Is tail recursive!

Call to length-1 in tail position

Structured Data

- A list is an example of a recursive data structure
  - Defined via a base case and inductive case:
    - A list is either the **empty list / null / '()**
    - Or a **cons cell** of any element and **another list**
- We can check whether it's `null?` or `cons?` or `list?`
- Can access via `car` and `cdr;` or `first` and `rest`
  - Many recursive functions on lists built using these

Exercise

Write a function to calculate the sum of a list

```
; (sum-list ‘(1 2)) is 3
(define (sum-list l)
  …)
```

Exercise

Write a function to calculate the sum of a list

```
; (sum-list '(1 2)) is 3
(define (sum-list l)
  …)
```

Answer (one of many)

```
(define (sum-list l)
  (if (eq? l '())
      0
      (+ (car l)
         (sum-list (cdr l)))))
```

Accumulator Passing

- Many functions can be written by *passing an **accumulator***: a value that is repeatedly extended to obtain a final value.
- Esp. in tail-recursive / looping algorithms; e.g.:

```
(define (sum-list l)
  (define (sum-loop l acc)
    (if (empty? l)
        acc
        (sum-loop (rest l)
                  (+ acc (first l)))))
  (sum-loop l 0))
```

S-exprs *(symbolic expressions)*

- The **S-expression** is our parenthesized notation for a list
  - Can use lists to group data common to some structure
- We can ***tag*** expressions with a symbol to note its "type"
  - `‘(point 2 3)`
  - `‘(square (point 0 1) 5)`
- Can define "constructor" functions

```
(define (mk-point x y)
  (list ‘point x y))
(define (mk-square pt0 len)
  (list ‘square pt0 len))
```

quasi-quotes

- Racket offers **quasi-quotes** to build S-expressions fast
- `` `(,x y 3) `` is equivalent to `` (list x `y `3) ``
    - I.e., Racket splices in values that are unquoted via `,`
    - (quasiquote …) will substitute any expression `,e` with the return value of `e` within the quoted S-expression
- Works multiple levels deep:
    - `` `(square (point ,x0 ,y0) (point ,x1 ,y1)) ``
- Can unquote entire expressions:
    - `` `(point ,(+ 1 x0) ,(- 1 y0)) ``

Exercise

Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list ‘point x y))
(define (mk-square pt0 pt1)
  (list ‘square pt0 pt1))
```

Exercise

Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list ‘point x y))
(define (mk-square pt0 pt1)
  (list ‘square pt0 pt1))
```

Answer

```
(define (mk-point x y)
  `(point ,x ,y))
(define (mk-square pt0 pt1)
  `(square ,pt0 ,pt1))
```

Pattern Matching

- Racket also has **pattern matching**
  - `(match e [pat0 body0] [pat1 body1]…)`
- Evaluates e and then checks each **pattern**, in order
- Pattern can bind variables, body can use pattern variables
- Many patterns (check docs to learn various useful forms)
- Patterns checked in order, first matching body is executed
  - Later bodies won't be executed, even if they also match!
- E.g.,

```
(match '(1 2 3)
       [`(,a ,b) b]
       [`(,a . ,b) b])  ; returns '(2 3)
```

Matching a literal

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

(binds n)

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches when e evaluates to some number?

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
   (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Never matches!

Subsumed by previous case!

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches a cons cell, binds x and y

```
(match e
  ['hello 'goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches a list of length three

Binds first element as `a0`, second as `a1`, etc…

Called a "quasi-pattern"

Can also test predicates on bound vars:

```
`(,(? nonnegative-integer? x) ,(? positive? y))
```

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)]
  [else 23])
```

Can also have a **default case** written via **else**

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string "error" if it is anything else

```
(define (foo x)
  (match x
    [(? …) …]
   …))
```

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

Answer (one of many)

```
(define (foo x)
  (match x
    [(? number? n) (* n 2)]
    [`(,a ,b ,_) `(,a ,b)]
    [else "error"]))
```

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string "error" if it is anything else

Answer (one of many)

Observe how quasipatterns and quasiquotes interact

```
(define (foo x)
  (match x
    [(? number? n) (* n 2)]
    [`(,a ,b ,_) `(,a ,b)]
    [else "error"]))
```

Structural Recursion

- **Structural recursion**
  - Recurs on some smaller piece of the input obtained by **destructing** (e.g., matching) on it.
- Easy to prove termination
  - Code is making input smaller at each recursive step, thus will eventually bottom out
- Much of the code you will write is structurally recursive
- But some things cannot be expressed in a structurally recursive way
  - E.g., *generative recursion*, other algorithms, …

Exercise

Consider that we define trees as follows:

```
(define (tree? t)
  (match t
    [`(leaf ,n) #t]
    [`(node ,(? tree? t0) ,(? tree? t1)) #t]
    [else #f]))
```

Assuming trees are sorted, write a recursive function using match patterns, `(least t)` to get the smallest element in the tree (i.e., bottom left leaf).

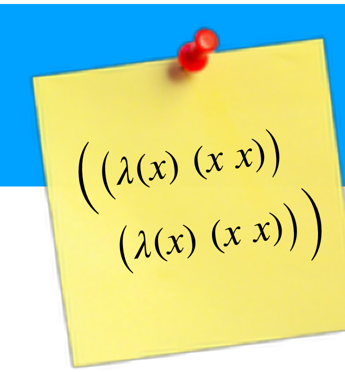
`(least (node (leaf 0) (leaf 1)))` should be `0`

(Hint: look at the definition of `tree?`)

Generative Recursion

- **Generative recursion**
  - Recurs on some structure built / calculated from input
- Not as easy (in general) to prove termination
  - How do we know it won't just loop forever?
- Strictly **more powerful** than structural recursion
  - Some programs we can't write w/ just structural recursion
  - E.g., QuickSort

Example

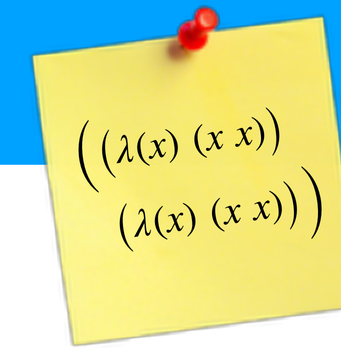


QuickSort is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l’ of the elements in l that are < p
- Also, construct l’’ of the elements in l that are > p
- Now, return…
  - QuickSort(l’) ++ [p] ++ QuickSort(l’’)

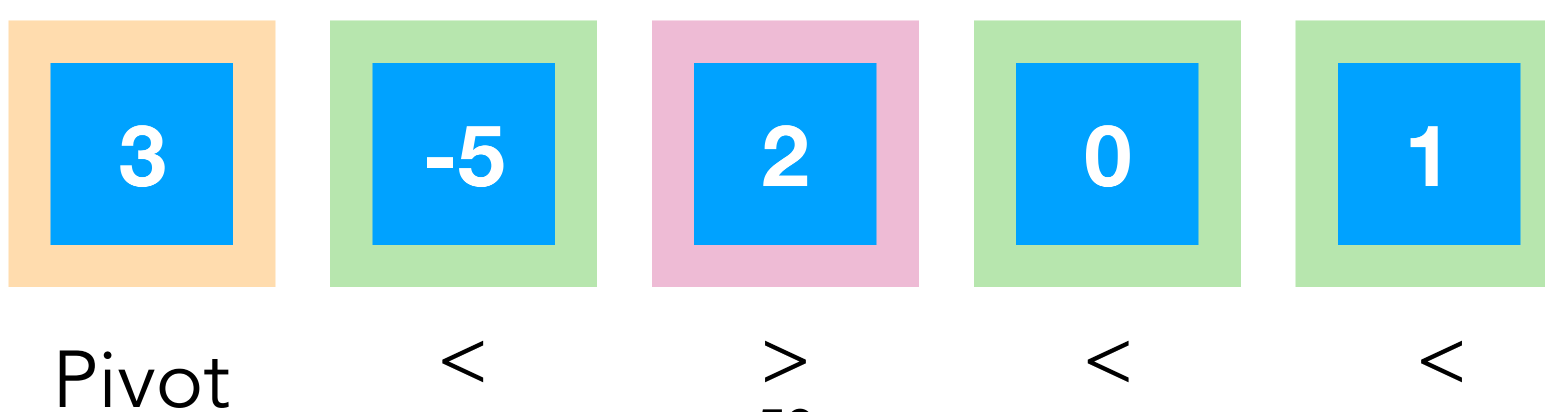
3	-5	2	0	1

Example

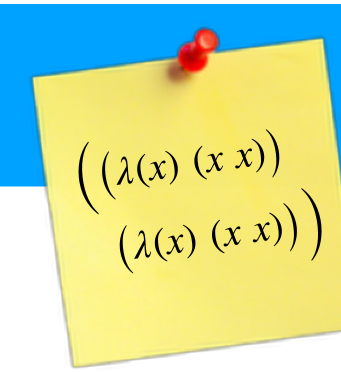


QuickSort is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l' of the elements in l that are < p
- Also, construct l'' of the elements in l that are > p
- Now, return…
  - QuickSort(l') ++ [p] ++ QuickSort(l'')



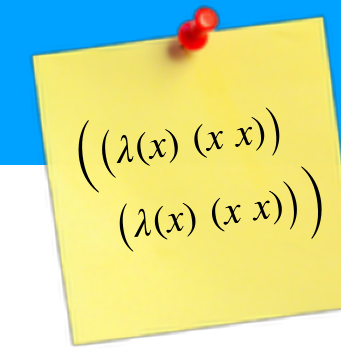
Example



QuickSort is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l’ of the elements in l that are < p
- Also, construct l’’ of the elements in l that are > p
- Now, return…
  - QuickSort(l’) ++ [p] ++ QuickSort(l’’)

Example



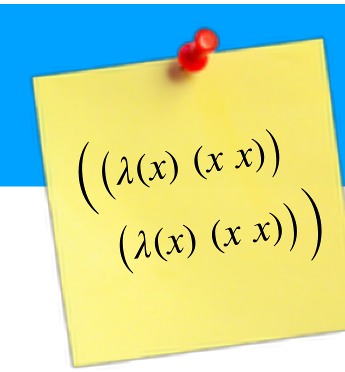
QuickSort is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l’ of the elements in l that are < p
- Also, construct l’’ of the elements in l that are > p
- Now, return…
  - QuickSort(l’) ++ [p] ++ QuickSort(l’’)

-5 0 1 3 2

Now sort these! Pivot >

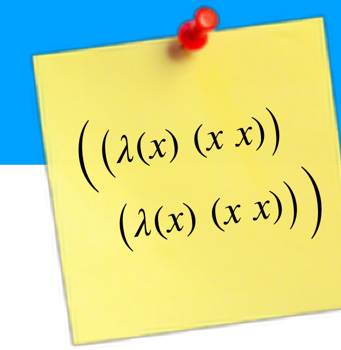
Example



QuickSort is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l' of the elements in l that are < p
- Also, construct l'' of the elements in l that are > p
- Now, return…
    - QuickSort(l') ++ [p] ++ QuickSort(l'')

Example

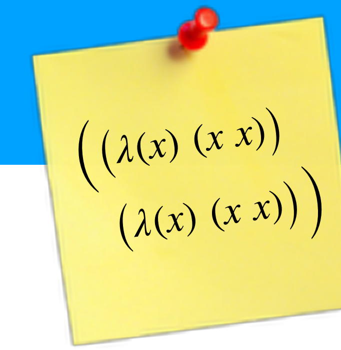


QuickSort is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l' of the elements in l that are < p
- Also, construct l'' of the elements in l that are > p
- Now, return…
  - QuickSort(l') ++ [p] ++ QuickSort(l'')

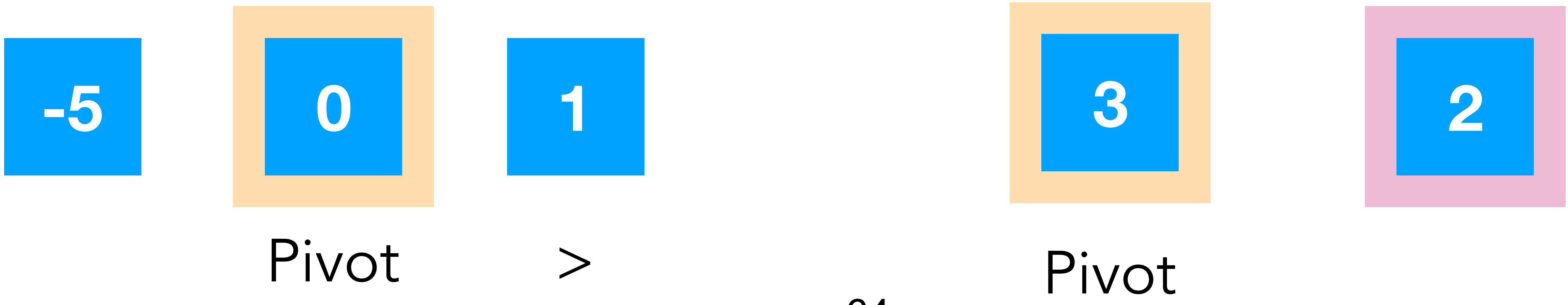
Now run quicksort on **these** Pivot >

Example

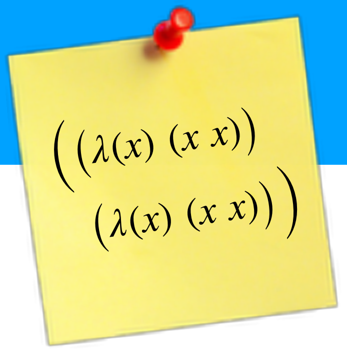


QuickSort is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l’ of the elements in l that are < p
- Also, construct l’’ of the elements in l that are > p
- Now, return…
  - QuickSort(l’) ++ [p] ++ QuickSort(l’’)



Example



QuickSort is a popular and fast sorting comparative sorting algorithm with O(n*log(n)) complexity

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l' of the elements in l that are < p
- Also, construct l'' of the elements in l that are > p
- Now, return…
  - QuickSort(l') ++ [p] ++ QuickSort(l'')

Now all sorted!

-5	0	1	3	2
			Original pivot	Just returns 2

Exercise

Write a function which returns the elements in a list, `l`, which are less than some number `n`

```
(define (elements< l n)
  …)
```

Hint: use `match`

Exercise

Write a function which returns the elements in a list, `l`, which are less than some number `n`

Answer (one of many)

```
(define (elements< l n)
  (match l
    ['() '()]
    [`(,first ,rest …)
      #:when (< first n)
     (cons first
           (elements< rest n))]
    [else (elements< (rest l) n)]))
```

Exercise

Can also easily write `elements>`

```
(define (elements< l n)
  (match l
    ['() '()]
    [`(,first ,rest ...) #:when (< first n)
      (cons first (elements< rest n))]
    [else (elements< (rest l) n)]))

(define (elements> l n)
  (match l
    ['() '()]
    [`(,first ,rest ...) #:when (> first n)
      (cons first (elements> rest n))]
    [else (elements> (rest l) n)]))
```

Redundant, will fix next week

Exercise

Complete the definition

- To sort list l, first choose a **pivot** element (arbitrary), p, from l
- Next, construct l’ of the elements in l that are < p
- Also, construct l’’ of the elements in l that are > p
- Now, return…
  - QuickSort(l’) ++ [p] ++ QuickSort(l’’)

```
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
             [restl (rest l)]
             [elements-lt (elements< restl pivot)]
             [elements-gt (elements> restl pivot)])
       …)))
```

Exercise

```
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
             [restl (rest l)]
             [elements-lt (elements< restl pivot)]
             [elements-gt (elements> restl pivot)])
        (append
         (quicksort elements-lt)
         (list pivot)
         (quicksort elements-gt)))))
```

Unfortunately, our implementation still has a bug!

Exercise

Exercise: find a list l such that

```
(not (equal? (sort l <) (quicksort l)))
```

```
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
             [restl (rest l)]
             [elements-lt (elements< restl pivot)]
             [elements-gt (elements> restl pivot)])
        (append
         (quicksort elements-lt)
         (list pivot)
         (quicksort elements-gt)))))
```

Exercise

Our QuickSort “drops” numbers

```
(not (equal? (sort ‘(1 1) <)
             (quicksort ‘(1 1))))
```

```
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
             [restl (rest l)]
             [elements-lt (elements< restl pivot)]
             [elements-gt (elements> restl pivot)])
        (append
         (quicksort elements-lt)
         (list pivot)
         (quicksort elements-gt)))))
```

Exercise

Solution is to make pivot a list!

```
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
             [pivot-list (elements= l pivot)]
             [restl (remove pivot l)]
             [elements-lt (elements< restl pivot)]
             [elements-gt (elements> restl pivot)])
        (append
         (quicksort elements-lt)
         pivot-list
         (quicksort elements-gt)))))
```

Exercise

Observe: QuickSort recursive on data **built from** input

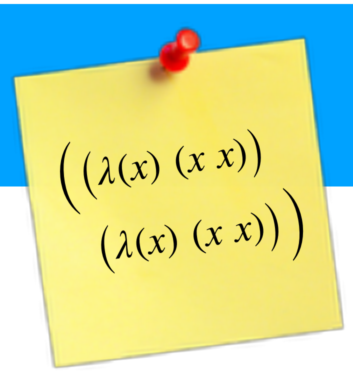
Thus, QuickSort uses **generative recursion**

```
(define (quicksort l)
  (if (empty? l)
      '()
      (let* ([pivot (first l)]
             [pivot-list (elements= l pivot)]
             [restl (remove pivot l)]
             [elements-lt (elements< restl pivot)]
             [elements-gt (elements> restl pivot)])
        (append
         (quicksort elements-lt)
         pivot-list
         (quicksort elements-gt)))))
```

Differential / Random Testing

- Want to be **very sure** our code is right
- One strategy: **fuzzing** (“fuzz testing”)
  - Generate huge amounts of input, throw it at our code
- One issue: need to check answer is correct
  - Idea one: compare against **known good** version
    - This is “differential” testing
    - Sometimes want a “slow” and “fast” version
      - Slow is obviously-correct but slow
  - Idea two: just check some **properties** of output
    - **Property-based** testing

Example

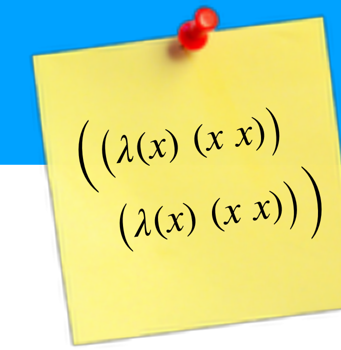


Let’s write a differential fuzzer for our QuickSort algorithm

```
(define (random-list i n)
  (if (= i 0)
      '()
      (cons (random 0 n)
            (random-list (- i 1) n))))
```

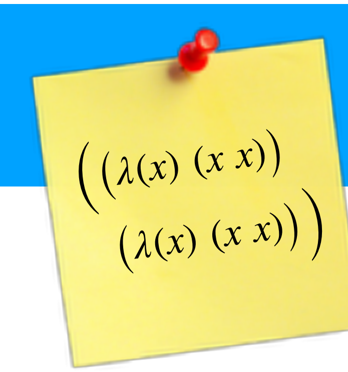
Generate random list of length `i`, whose elements are all in `[0,n-1]`

Example




```
(define (counterexamples num-tries list-size max-n)
  (define (loop i l)
    (if (= i 0)
        l
        (let* ([lst (random-list list-size max-n)]
               [sorted-via-sort (sort lst <)]
               [sorted-via-qsort (quicksort lst)])
          (if (equal? sorted-via-sort sorted-via-qsort)
              (loop (- i 1) l)
              (loop (- i 1) (cons lst l))))))
  (loop num-tries '()))
```

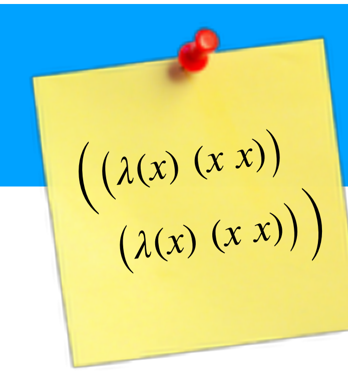
Example



Compare our `quicksort` against Racket's `sort`

```
(define (counterexamples num-tries list-size max-n)
  (define (loop i l)
    (if (= i 0)
        l
        (let* ([lst (random-list list-size max-n)]
               [sorted-via-sort (sort lst <)]
               [sorted-via-qsort (quicksort lst)])
          (if (equal? sorted-via-sort sorted-via-qsort)
              (loop (- i 1) l)
              (loop (- i 1) (cons lst l))))))
  (loop num-tries '()))
```

Example




```
(define (counterexamples num-tries list-size max-n)
  (define (loop i l)
    (if (= i 0)
        l
        (let* ([lst (random-list list-size max-n)]
               [sorted-via-sort (sort lst <)]
               [sorted-via-qsort (quicksort lst)])
          (if (equal? sorted-via-sort sorted-via-qsort)
              (loop (- i 1) l)
              (loop (- i 1) (cons lst l))))))
  (loop num-tries '()))
```

```
(counterexamples 300 300 1000)
```