CS 4
Fundamentals of Computer Programming

Lecture 3: January 9, 2015

Recursion and Iteration

```
(define (f)
  (display "Home")
  (display "Sweet")
  (f))
```

```
10 Print "Home"
20 Print "Sweet"
30 GO TO 10
```

Last time

- The substitution model
- Boolean values (`#t`, `#f`)
- The `if` special form

Today

- Recursively-defined procedures
- Recursion and the substitution model
- Proving correctness
- Iterative procedures
- Linear recursion
- Tree recursion

Math Problem:

- How do I turn a repeating decimal into a rational number?
- (*e.g.*, 0.666666666666666666666…)

Repeating Fraction

- x = 0.666666666666666666666…
- 10x = 6.666666666666666666….
- 10x = 6 + 0.666666666666666666…
- 10x = 6 + x
- 9x = 6
- x = 6/9 = 2/3

Question #2

- What is the value of:

  1 + 1/2 + 1/4 + 1/8 + …?

- x = 1 + 1/2 + 1/4 + 1/8 + …
- x = 1 + 1/2 * (1 + 1/2 + 1/4 + 1/8 + …)
- x = 1 + 1/2 * x
- x = 1 + x/2
- x/2 = 1
- x = 2

The trick (or technique)

- "Trick" in both of these:
  - Turn the problem into a **recursive** definition
  - Definition "refers back to itself"
    - $10x = 6 + x$
    - $x = 1 + x/2$

Summing integers

- Let's use this approach to solve the problem: How do we compute the sum of the first N integers?
  - (Where N is a non-negative integer)

Decomposing the sum

- Sum of first N integers =
  - N + N-1 + N-2 + … + 1
  - N + (N-1 + N-2 + … + 1)
  - N + [sum of first N-1 integers]

to Scheme

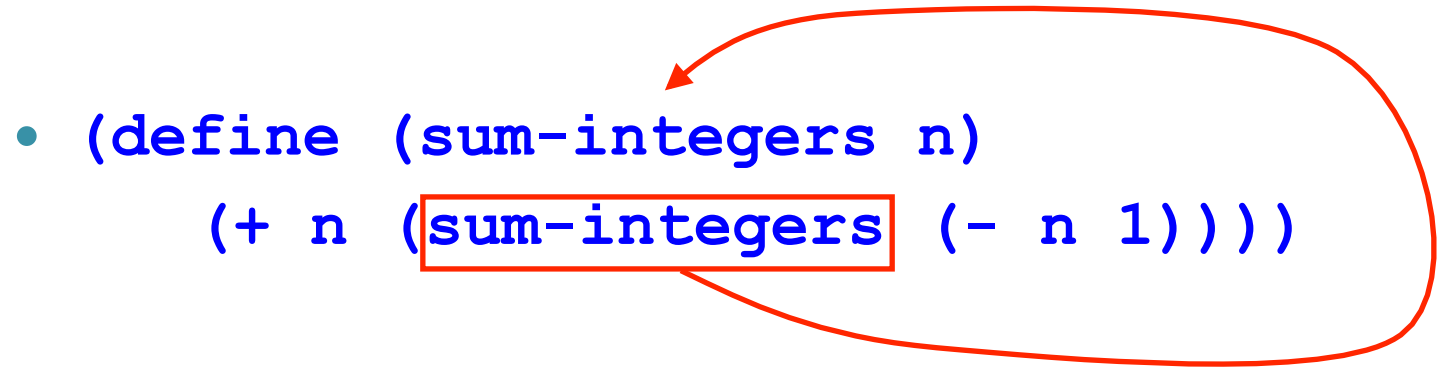
- Sum of first N integers =

  N + [sum of first N-1 integers]

- Convert to Scheme (first attempt):

```
(define (sum-integers n)
    (+ n (sum-integers (- n 1))))
```

Recursion in Scheme



```
(define (sum-integers n)
    (+ n (sum-integers (- n 1))))
```

- **`sum-integers`** is defined in terms of *itself*
- This is a *recursively defined* procedure

  ... which happens to be incorrect
- What's the error?

Almost…

- What's wrong?

```
(define (sum-integers n)
  (+ n (sum-integers (- n 1))))
```

- Gives us:

N + N-1 + …+ 1 + 0 + -1 + -2 + …

Debugging

- Fixing the problem:
  - it doesn't stop at zero!
- Revised:

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

- How does this evaluate?

Evaluating

- Evaluate: `(sum-integers 3)`
  - evaluate `3` → `3`
  - evaluate `sum-integers` →

```
(lambda (n) (if ...))
```

  - apply `(lambda (n)`

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

 to `3`
    - substitute `3` for `n` in `lambda` expression to get:

→ `(if (= 3 0) 0`

```
      (+ 3 (sum-integers (- 3 1))))
```

Evaluating…

- Evaluate: `(if (= 3 0) 0 (+ 3 (sum-integers (- 3 1))))`
  - evaluate `(= 3 0)`
    - evaluate `3` → `3`
    - evaluate `0` → `0`
    - evaluate `=` → `=`
    - apply `=` to `3`, `0` → `#f`
  - since expression is false, replace with false clause:
    `(+ 3 (sum-integers (- 3 1)))`
  - evaluate: `(+ 3 (sum-integers (- 3 1)))`

Evaluating…

- Evaluate `(+ 3 (sum-integers (- 3 1)))`
  - evaluate `3` → `3`
  - evaluate `(sum-integers (- 3 1))`
    - evaluate `(- 3 1)` ... [skip steps] … → `2`
    - evaluate `sum-integers` →

```
(lambda (n) (if ...))
```

`Note: now pending: (+ 3 ...)`

Evaluating…

- Apply `(lambda (n)`
  `(if (= n 0)`
  `0`
  `(+ n (sum-integers (- n 1))))))` to `2`
  - substitute `2` for `n` in `lambda` expression to get:

→ `(if (= 2 0) 0`
`(+ 2 (sum-integers (- 2 1)))`

Evaluating…

- Evaluate: `(if (= 2 0) 0`
  `(+ 2 (sum-integers (- 2 1)))`
  - evaluate `(= 2 0)` … [skip steps] … → `#f`
  - since expression is false, replace with false clause:
  - `(+ 2 (sum-integers (- 2 1)))`
  - evaluate: `(+ 2 (sum-integers (- 2 1)))`

Evaluating…

- Evaluate `(+ 2 (sum-integers (- 2 1)))`
  - evaluate `2` → `2`
  - evaluate `(sum-integers (- 2 1))`
    - evaluate `(- 2 1)` ... [skip steps] … → `1`
    - evaluate `sum-integers` →

```
(lambda (n) (if ...))
```

    - apply `(lambda (n) ...)` to `1`, subst `1` for `n`...
    -
```
(if (= 1 0) 0
    (+ 1 (sum-integers (- 1 1))))
```

      - evaluate `(= 1 0)` ... [skip steps] … → `#f`
      - Therefore, evaluate false clause:
      - `(+ 1 (sum-integers (- 1 1)))`

Evaluating…

- Evaluate `(+ 1 (sum-integers (- 1 1)))`
  - evaluate `1` → `1`
  - evaluate `(sum-integers (- 1 1))`
    - evaluate `(- 1 1)` ... [skip steps] … → `0`
    - evaluate `sum-integers` →
      `(lambda (n) (if ...))`
    - apply `(lambda (n) ...)` to `0`, subst `0` for `n` ...
    - `(if (= 0 0) 0`
      `(+ 1 (sum-integers (- 0 1))))`
      - evaluate `(= 0 0)` → `#t`
      - result: `0` (true clause)

Pending: `(+ 3 (+ 2 (+ 1 0)))`

Evaluating

- Now we've evaluated all the `if` expressions
- Finish evaluating pending expressions:
- Evaluate `(+ 1 0)`
  - Usual steps ... [skip steps] … → `1`
- Evaluate `(+ 2 1)`
  - Usual steps ... [skip steps] … → `3`
- Evaluate `(+ 3 3)`
  - Usual steps ... [skip steps] … → `6`
- Final result: `6`

Whew!!!

- The substitution model works fine for recursion
- Recursive calls are well-defined
- Careful application of the model shows us what they mean and how they work

Revisited (summarizing steps)

- Evaluate: `(sum-integers 3)`
- `(if (= 3 0) 0 (+ 3 (sum-integers (- 3 1))))`
- `(if #f 0 (+ 3 (sum-integers (- 3 1))))`
- `(+ 3 (sum-integers (- 3 1)))`
- `(+ 3 (sum-integers 2))`
- `(+ 3 (if (= 2 0) 0 (+ 2 (sum-integers (- 2 1)))))`
- `(+ 3 (+ 2 (sum-integers 1)))`
- `(+ 3 (+ 2 (if (= 1 0) 0`
  `(+ 1 (sum-integer (- 1 1))))))`
- `(+ 3 (+ 2 (+ 1 (sum-integers 0))))`
- `(+ 3 (+ 2 (+ 1 (if (= 0 0) 0 ...))))`
- `(+ 3 (+ 2 (+ 1 (if #t 0 ...))))`
- `(+ 3 (+ 2 (+ 1 0)))` → `[obvious steps]` → 6

Very important *design pattern*!

```
(define (sum-integers n)
  (if (= n 0)   ;; test
      0         ;; base case
      (+ n (sum-integers (- n 1))))) ;; recursion
```

- We will see this pattern over and over again as the course progresses

Key idea

- You have a large (potentially unbounded) problem to solve…
- Figure out how to **reduce** it to a *smaller* problem with a constant amount of work
- Solve the smaller problem, and **combine** its result with available data to solve the larger problem
- Eventually, handle the smallest case (**base case**)

Design process

- In practice, usually we *first* figure out how to handle the base case (or cases)
- Then we figure out how to decompose problem into smaller problem of similar kind
  - to be solved recursively
- Then we figure out how to combine the solution of the smaller problem with the current value to get final answer
- Then we're done!

Proving correctness

- Actually, not quite done...
- We have our function, but haven't *proved* that it is correct
- Proving correctness is very hard in general...
- ...but with recursive procedures it can often be very easy!
  - at least to get informal proof

Recall math proofs

- Prove that something is true
  - *e.g.* `(sum-integers n)` = n*(n+1)/2
  - Can verify for fixed n:
    - N = 0: 0*(0+1)/2 = 0
    - N = 1: 1*(1+1)/2 = 1
    - N = 2: 2*(2+1)/2 = 3
    - N = 3: 3*(3+1)/2 = 6
    - ….
  - But when N is *unbounded*…
    - how do we know this always holds?

Proof by induction

- Mathematical induction:

  In order to prove that something is true

  - *e.g.* `(sum-integers n)` = n*(n+1)/2
  - show that it's true for the base case:
    - N = 0,
      - here: 0*(0+1)/2 = 0
  - assume it holds true for some arbitrary N
  - show that it must hold true for N+1

True for N+1?

Show that it holds true for N + 1:

- sum-integers(N + 1) = sum-integers(N) + (N + 1)
- = (N*(N + 1) / 2) + (N + 1)
- = (N*(N + 1) + 2*(N + 1)) / 2
- = ((N + 1) * (N + 2)) / 2
- *Let K = N + 1:*
- = (K * (K + 1)) / 2
- QED

Inductive Ladder

- Induction:
  - show how to solve the base case
  - show how to prove each successive case given the previous case
  - climb the ladder to see will be true for any N

In a similar manner…

- Can argue correctness of:

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

- Show that the procedure is correct for some small `n` (base case)
- Argue that `n+1` case is correct based on correctness of `n` case

Base case

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

- if `n` = 0, `(sum-integers n)` = 0
- obviously correct

Recursive step

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

- If `(sum-integers (- n 1))` correct, then
- `(+ n (sum-integers (- n 1)))` also must be correct
  - by definition of what the sum of the integers up to `n` means

Conclusion

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

- All the cases are correct
- Therefore we have *proven* that **`sum-integers`** is correct!

Decomposition

- Decomposition is a powerful technique
  - take a large problem
  - reduce it to smaller pieces
  - AKA "divide and conquer"
- We'll see many ways to decompose problems…
  - many will use recursion

Now, let's think about:

- How do our computations unfold?
  - in space
  - in time
- This will lead us to consider:
- efficiency of program execution

  (first look at a huge subject)

Evolution of computation

- `(sum-integers 3)`
- `(+ 3 (sum-integers 2))`
- `(+ 3 (+ 2 (sum-integers 1)))`
- `(+ 3 (+ 2 (+ 1 (sum-integers 0))))`
- `(+ 3 (+ 2 (+ 1 0)))`

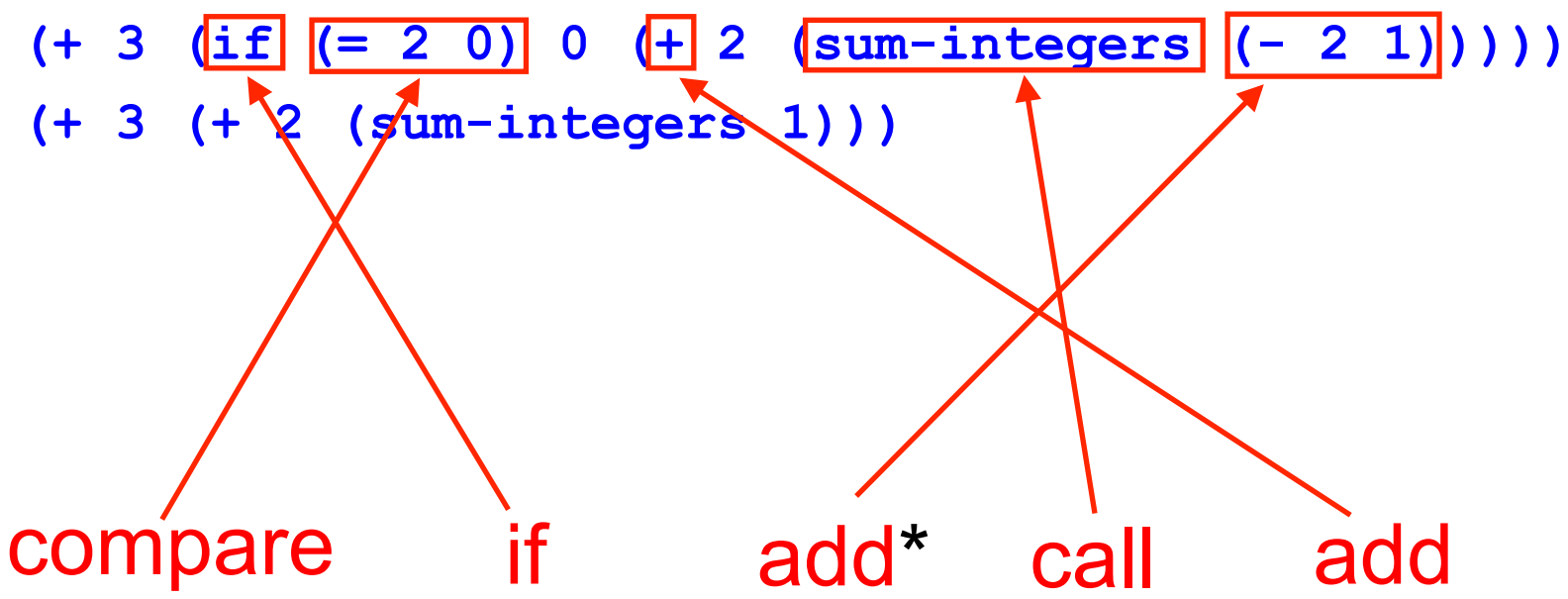
Efficiency of `sum-integers`

```
(sum-integers 3)
(+ 3 (sum-integers 2))
(+ 3 (+ 2 (sum-integers 1)))
(+ 3 (+ 2 (+ 1 (sum-integers 0))))
(+ 3 (+ 2 (+ 1 0)))
```

- On input `n`:
  - How many calls to `sum-integers`?
    - (`n` + 1)
  - How much "work" per call?

Work per call?

```
(+ 3 (sum-integers 2))
(+ 3 (if (= 2 0) 0 (+ 2 (sum-integers (- 2 1)))))
(+ 3 (+ 2 (sum-integers 1)))
```



*(subtract takes the same amount of time as add)

Work per call?

```
(sum-integers 3)
(+ 3 (sum-integers 2))
(+ 3 (+ 2 (sum-integers 1)))
(+ 3 (+ 2 (+ 1 (sum-integers 0))))
(+ 3 (+ 2 (+ 1 0)))
```

- On input **`n`**:
- How many calls to **`sum-integers`**?
  - **`n+1`**
- How much work per call?
  - constant
  - one comparison, one **`if`**, two additions, one function call (at most)
- How many deferred operations at end?
  - **`n`**

How long does it run?

On input `n`:

- How many calls to `sum-integers`?
  - `n+1`
- How much work per call?
  - constant (one comparison, one if, two additions, one function call)
- time = $(N+1) * (T_{cmp} + T_{if} + 2*T_{add} + T_{call})$
- time = $C_1 + N*C_2$
  - for some constants C_1 and C_2

Linear recursive processes

```
(sum-integers 3)
(+ 3 (sum-integers 2))
(+ 3 (+ 2 (sum-integers 1)))
(+ 3 (+ 2 (+ 1 (sum-integers 0))))
(+ 3 (+ 2 (+ 1 0)))
```

- time = $C_1 + N*C_2$
- This is what we call a **linear recursive** process
- Makes **linear** number of function calls
  - *i.e.* proportional to `n`
- Keeps a chain of **deferred** operations linear in size w.r.t. input
  - *i.e.* proportional to `n`

Linear recursive processes

- To compute an answer using a linear recursive process, you need
  - an amount of time which is linear in the size of the input
    - to execute all the function calls
  - an amount of space which is linear in the size of the input
    - to hold the data of the deferred computations

Another strategy

To sum integers…

add up as we go along:

- 1 2 3 4 5 6 7 …
- 1 1+2=3 1+2+3=6 1+2+3+4=10 …
- 1 3 6 10 15 21 28 …

Alternate definition

```
(define (sum-int n)
  (sum-iter 0 n 0))   ;; start at 0, sum is 0

(define (sum-iter current max sum) ;; "helper" function
  (if (> current max)
      sum
      (sum-iter (+ 1 current) ;; recursive call
                max
                (+ current sum))))
```

Evaluation of `sum-int`

```
(define (sum-int n)
  (sum-iter 0 n 0))

(define (sum-iter current max sum)
  (if (> current max)
      sum
      (sum-iter (+ 1 current) max (+ current sum))))
```

- `(sum-int 3)`
- `(sum-iter 0 3 0)`
- `(if (> 0 3) ...)`
- `(sum-iter 1 3 0)`
- `;; etc.`

Evaluation of `sum-int`

```
(define (sum-iter current max sum)
  (if (> current max)
      sum
      (sum-iter (+ 1 current) max (+ current sum))))
```

- `(sum-iter 0 3 0)`
- `(if (> 0 3) ...)`
- `(sum-iter 1 3 0)  ;; 1+0=1, 0+0=0`

Evaluation of `sum-int`

```
(define (sum-iter current max sum)
  (if (> current max)
      sum
      (sum-iter (+ 1 current) max (+ current sum))))
```

- `(sum-iter 0 3 0)`
- `(if (> 0 3) ...)`
- `(sum-iter 1 3 0)`
- `(if (> 1 3) ...)`
- `(sum-iter 2 3 1)` `;; 1+1=2, 1+0=1`

Evaluation of `sum-int`

```
(define (sum-iter current max sum)
  (if (> current max)
      sum
      (sum-iter (+ 1 current) max (+ current sum))))
```

- `(sum-iter 0 3 0)`
- `(if (> 0 3) ...)`
- `(sum-iter 1 3 0)`
- `(if (> 1 3) ...)`
- `(sum-iter 2 3 1)`
- `(if (> 2 3) ...)`
- `(sum-iter 3 3 3)` `;; 1+2=3, 2+1=3`

Evaluation of `sum-int`

- `...`
- `(sum-iter 2 3 1)`
- `(if (> 2 3) ...)`
- `(sum-iter 3 3 3)`
- `(if (> 3 3) ...)`
- `(sum-iter 4 3 6)` `;; 1+3=4, 3+3=6`

Evaluation of `sum-int`

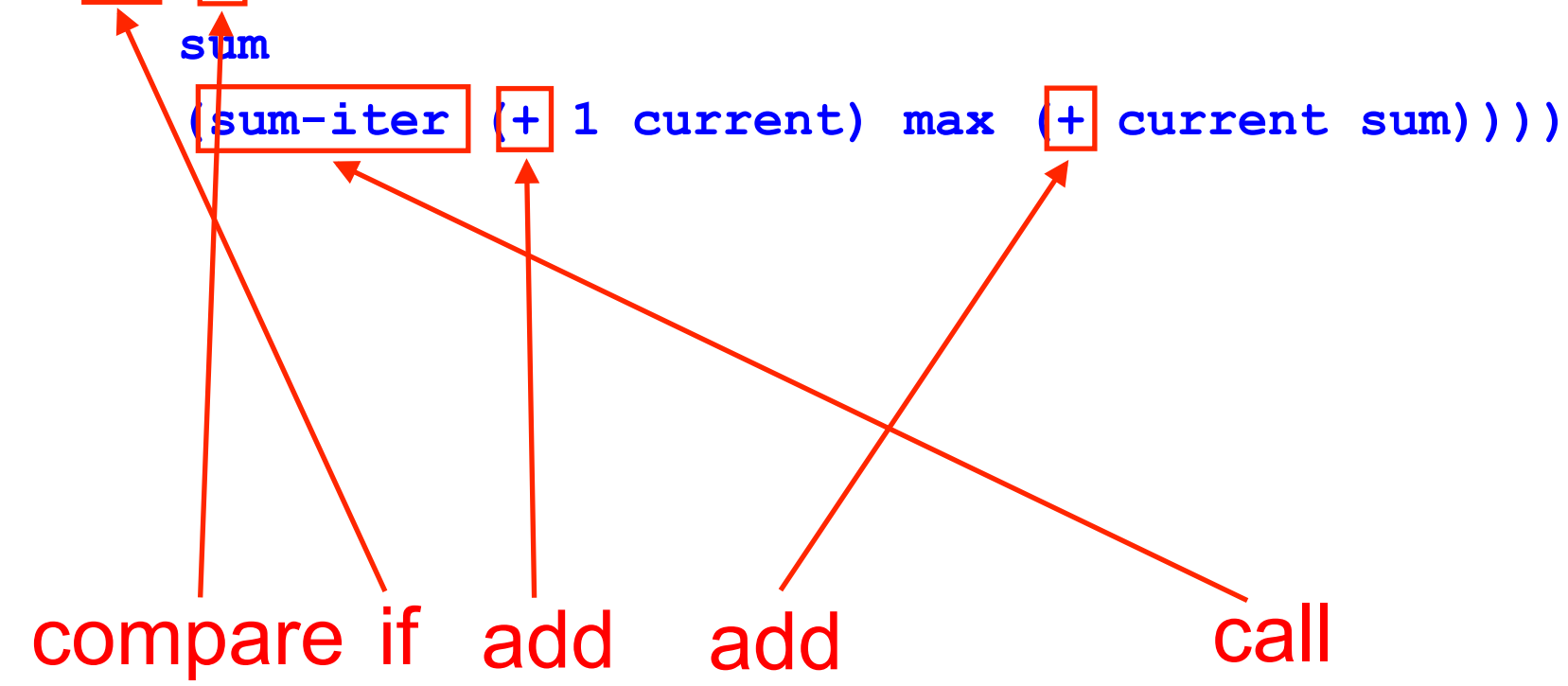
- `...`
- `(sum-iter 2 3 1)`
- `(if (> 2 3) ...)`
- `(sum-iter 3 3 3)`
- `(if (> 3 3) ...)`
- `(sum-iter 4 3 6)`
- `(if (> 4 3) 6 ...) ;; sum = 6`
- `(if #t 6 ...)`
- Answer: 6

Efficiency of `sum-int`

- `(sum-int 3)`
- `(sum-iter 0 3 0)`
- `(sum-iter 1 3 0)`
- `(sum-iter 2 3 1)`
- `(sum-iter 3 3 3)`
- `(sum-iter 4 3 6)`
- `6`
- On input **`n`**:
- How many calls to **`sum-iter`**?
  - **`n+2`**
- How much work per call?

`sum-iter`: work per call

```
(define (sum-iter current max sum)
  (if (> current max)
      sum
      (sum-iter (+ 1 current) max (+ current sum))))
```



Efficiency of `sum-int`

- On input `n`:
- How many calls to `sum-iter`? (`n+2`)
- How much work per call?
  - constant
  - one comparison, one if, two additions, one call
- How many deferred operations?
  - none!

How long does it run?

On input `n`:

- How many calls to `sum-iter`?
  - `n+2`
- How much work per function call?
  - constant
  - one comparison, one `if`, two additions, one function call

- time = $T_{sum\text{-}int} + (N+2) * T_{sum\text{-}iter}$
- $T_{sum\text{-}iter} = T_{cmp} + T_{if} + 2*T_{add} + T_{call}$
- $T_{sum\text{-}int} = T_{call}$
- time = $T_{call} + (N+2)*(T_{cmp}+2*T_{add}+T_{if}+T_{call})$
- time = $C_1 + N*C_2$

What's different about these computations?

Old

```
(sum-integers 3)
(+ 3 (sum-integers 2))
(+ 3 (+ 2 (sum-integers 1)))
(+ 3 (+ 2 (+ 1 (sum-integers 0))))
(+ 3 (+ 2 (+ 1 0)))
```

New

```
(sum-int 3)
(sum-iter 0 3 0)
(sum-iter 1 3 0)
(sum-iter 2 3 1)
(sum-iter 3 3 3)
(sum-iter 4 3 6)
```

Linear iterative processes

- `(sum-int 3)`
- `(sum-iter 0 3 0)`
- `(sum-iter 1 3 0)`
- `(sum-iter 2 3 1)`
- `(sum-iter 3 3 3)`
- `(sum-iter 4 3 6)`
- `6`
- time = $C_1 + N*C_2$

- This is what we call a **linear iterative** process
- Makes linear # calls
- State of computation kept in a constant # of state variables
  - state does not grow with problem size
  - requires a constant amount of storage space

Linear recursive vs linear iterative

- Both require computational *time* which is linear in size of input
- Linear recursive process requires *space* that is also linear in size of input
  - why?
- Linear iterative process requires *constant* space
  - *not* proportional to size of input
  - more space-efficient than linear recursive process

Linear recursive vs linear iterative

- Why not always use linear iterative instead of linear recursive algorithms?
  - often the case in practice
- Recursive algorithm sometimes much easier to write
  - and to prove correct!
- Sometimes space efficiency isn't the limiting factor

Helper functions

- Linear iterative functions often use "helper functions"
  - functions only intended to be called by the main function we're interested in
- Helper functions have extra arguments
  - which provide the extra state variables we need
- Here, **`sum-int`** is the function we want to define
- **`sum-iter`** is the helper function
- Later, we'll see how to put helper functions "inside" of functions that use them

Helper functions

- Helper functions may seem weird to programmers coming from imperative languages
- Why not just use a variable and change its value inside a loop?
- Answer: Because we don't have variables or looping constructs yet!
  - (Will see them later)
- Interesting point: often don't *need* variables/ loops to compute many functions of interest

A note on abuse of terminology

- We say "linear recursive" and "linear iterative" *processes* even though both use recursive *procedures*
- This is the book's terminology
  - kind of confusing
- In this sense, they mean that a recursive *process* involves pending operations, while an iterative *process* doesn't

Fibonacci numbers

- Familiar sequence?
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, ???
- What's the rule for this sequence?
  - fib(0) = 0
  - fib(1) = 1
  - fib(n) = fib(n-1) + fib(n-2)

Simple Scheme translation

- What's the rule for this sequence?
  - fib(0) = 0
  - fib(1) = 1
  - fib(n) = fib(n-1) + fib(n-2)

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

Is this correct?

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

Is this correct?

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

- Base cases (`0` and `1`) ... check

Is this correct?

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

- Base cases (`0` and `1`) ... check
- Recursive case ... check

Is this correct?

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

- Base cases (`0` and `1`) ... check
- Recursive case ... check
- No other cases, so this must be correct!

What's different?

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

What's different?

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

- `fib` has *two* recursive calls for each evaluation

Evolution of `fib`

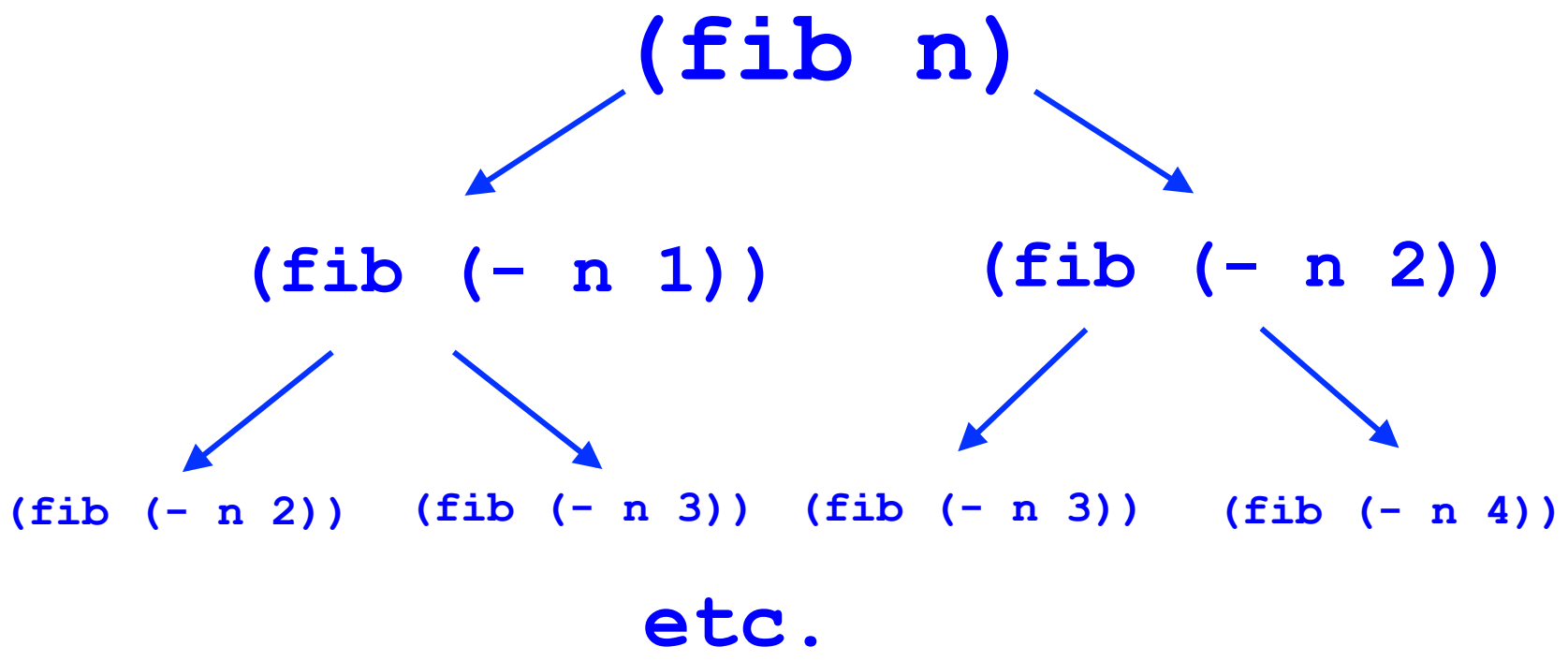
```
(fib 5)
(+ (fib 4) (fib 3))
(+ (+ (fib 3) (fib 2)) (+ (fib 2) (fib 1)))
(+ (+ (+ (fib 2) (fib 1)) (+ (fib 1) (fib 0))) (+ (+ (fib 1) (fib 0)) 1))
(+ (+ (+ (+ (fib 1) (fib 0)) 1) (+ 1 0))) (+ (+ 1 0) 1))
(+ (+ (+ (+ 1 0) 1) (+ 1 0)) (+ (+ 1 0) 1) ;; only base cases
(+ (+ (+ 1 1) 1) (+ 1 1))
(+ (+ 2 1) 2)
(+ 3 2)
5
```

Evolution of `fib`

- First call
  - calls 2 `fib`s
    - each calls 2 more `fib`s (for total of 4)
      - each calls 2 more (for total of 8)
        - ….until hit base cases

`fib` evolution

- Expands into a “tree” instead of linear chain



Tree recursive processes

```
(fib 5)
→ (+ (fib 4)  (fib 3))
→ (+ (+ (fib 3) (fib 2))(+ (fib 2) (fib 1)))
→ ...
```

- This is what we call a **tree recursive** process
- Calls fan out in a tree
- How many calls to `fib`?
  - *exponential* in size of argument
  - (not perfectly "balanced" tree)

How long?

- $T(\text{fib } n) = T_{fib} + T(\text{fib } n-1) + T(\text{fib } n-2)$
- where T_{fib} is the time spent inside one `fib` call
  - just a sum of constant factors as before

How long?

- $T(\text{fib } n) = T_{fib} + T(\text{fib } n-1) + T(\text{fib } n-2)$
- and $T(\text{fib } n-1) >= T(\text{fib } n-2)$, so simplifying:
- $T(\text{fib } n) <= T_{fib} + 2 * T(\text{fib } n-1)$
- $T(\text{fib } n) <= T_{fib} + 2 * (T_{fib} + 2 * T(\text{fib } n-2))$
- $T(\text{fib } n) <= T_{fib} + 2*T_{fib} + 4*T_{fib} + \ldots$
- $T(\text{fib } n) <= T_{fib} * 2^n * (1 + ½ + ¼ + \ldots)$
- $T(\text{fib } n) <= T_{fib} * 2^{(n+1)}$

How long?

- $T(\text{fib } n) <= T_{fib} * 2^{(n+1)}$
- Is that fast?
- No!
- As input size increase, time required gets *exponentially* larger
- AKA a "bad thing"

Caveat

- $T(\text{fib } n) <= T_{fib} * 2^{(n+1)}$
- Technically, this says that **`fib`** has an exponential *upper bound*
- Can also prove that **`fib`** has an exponential *lower bound*, but it's harder
  - see book for more on this
- Point: **`fib`** has exponential time requirements

Reason?

- *Why* is recursive **`fib`** so slow?
- Have to recompute values have already computed before, again and again and again
  - **`(fib 5)`** → **`(fib 2)`** computed 3 times
  - **`(fib 10)`** → **`(fib 3)`** computed 21 times
- Straightforward recursive algorithm for **`fib`**
  - easy to prove correct (good)
  - unacceptably slow (bad)

Summary

- Processes involving unbounded numbers of computations can be represented as recursive computations
- Recursion can be understood using the substitution model
- Recursive procedures are designed using a base case/recursive case pattern
- Recursive *procedures* generate different kinds of computational *processes* (linear recursive, linear iterative, tree recursive)

Next time

- Finish discussion of how processes unfold
- Show how to quantify the efficiency of processes (asymptotic complexity)