SICP Chapter 1 Answers

ProducerMatt

August 27, 2022

Contents

1 HOW THIS DOCUMENT IS MADE

TODO

```
(define (foo a b)
  (+ a (* 2 b)))

(foo 5 3)
```

^ Dynamically evaluated when you press "enter" on the `BEGIN_SRC` block!

1.0.1 Also consider:

- `:results output` for what the code prints
- `:exports code` or `:exports results` to just get one or the other

$a + (\pi \times b)$ <~ inline Latex btw :)

1.0.2 Current command for conversion

```
pandoc --from org --to latex 1.org -o 1.tex -s; xelatex 1.tex
```

1.1 Helpers for org-mode tables

1.1.1 `try-these`

Takes function `f` and list `testvals` and applies `f` to each item `i`. For each `i` returns a list with `i` and the result. Useful dor making tables with a column for input and a column for output.

```
;; Surely this could be less nightmarish
(define (try-these f . testvals)
  (let ((l (if (and (= 1 (length testvals))
                    (list? (car testvals)))
               (car testvals)
               testvals)))
    (map (λ (i) (cons i
                      (cons (if (list? i)
                                (apply f i)
                                (f i))
                            #nil)))
         l)))
```

1.1.2 `transpose-list`

"Rotate" a list, for example from `'(1 2 3)` to `'('(1) '(2) '(3))`

```
(define (transpose-list l)
  (map list l))
```

1.1.3 `print-as-rows`

For manually printing items in rows to stdout. Not currently used.

```
(define (p-nl a)
  (display a)
  (newline))
```

```
(define (print-spaced args)
  (let ((a (car args))
        (d (cdr args)))
    (if (null? d)
        (p-nl a)
        (begin (display a)
               (display " ")
               (print-spaced d)))))
(define (print-as-rows . args)
  (let ((a (car args))
        (d (cdr args)))
    (if (list? a)
        (if (= 1 (length args))
            (apply print-as-rows a)
            (print-spaced a))
        (p-nl a))
    (if (null? d)
        '()
        (apply print-as-rows d))))
```

1.1.4 `print-table`

Print `args` as a table separated by pipes. Optionally print spacer for colnames.

```
(use-modules (ice-9 format))
(define* (print-table table #:key (colnames #f) (mode #f))
  (let ((fmtstr
         (cond ((or (eq? mode #f)
                    (equal? mode "display")
                    (equal? mode "~a"))
                " ~a |")    ;; print objects for human viewing
               ((or (eq? mode #t)
                    (equal? mode "write")
                    (equal? mode "~s"))
                " ~s |")))) ;; print objects for correctly (read)ing
↪ back
    (define* (print-row ll #:key (fmt fmtstr))
      (format #t "~&|")
      (map (λ(x) (format #t fmt x)) ll)
      (format #t "~%"))
    (define (iter t)
      (print-row (car t))
      (if colnames
          (print-row (car t) #:fmt "---|"))
      (map print-row (cdr t)))
    (cond ((and (= 1 (length table))
```

```
                (list? (car table))) (iter (car table)))
          ((<= 1 (length table)) (iter table))
          (else error "Invalid Input??"))))
```

```
<<print-table>>
(let* ((l (iota 3))
      (table (list
              (list 'column-1 'column-2 'column-3 'column-4)
              (cons 'row-a l)
              (cons 'row-b l)
              (cons 'row-c l))))
  (print-table table #:colnames #t ))
```

2 Exercise 1.1

2.1 Question

Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

2.2 Answer

```
10 ;; 10
(+ 5 3 4) ;; 12
(- 9 1) ;; 8
(/ 6 2) ;; 3
(+ (* 2 4) (- 4 6)) ;; 6
(define a 3) ;; a=3
(define b (+ a 1)) ;; b=4
(+ a b (* a b)) ;; 19
(= a b) ;; false
(if (and (> b a) (< b (* a b)))
    b
    a) ;; 4
(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25)) ;; 16
(+ 2 (if (> b a) b a)) ;; 6
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1)) ;; 16
```

3 Exercise 1.2

3.1 Question

Translate the following expression into prefix form:

$$\frac{5 + 2 + (2 - 3 - (6 + \frac{4}{5}))}{3(6 - 2)(2 - 7)}$$

3.2 Answer

```
(/ (+ 5 2 (- 2 3 (+ 6 (/ 4 5))))
   (* 3 (- 6 2) (- 2 7)))
```

4 Exercise 1.3

4.1 Text

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

4.2 Question

Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

4.3 Answer

```
<<square>>
(define (sum-square x y)
  (+ (square x) (square y)))
(define (square-2of3 a b c)
  (cond ((and (>= a b) (>= b c)) (sum-square a b))
        ((and (>= a b) (> c b)) (sum-square a c))
        (else (sum-square b c))))
```

```
<<EX1-3>>
<<try-these>>
 (try-these square-2of3 '(7 5 3)
                        '(7 3 5)
                        '(3 5 7))
```

5 Exercise 1.4

5.1 Question

Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

5.2 Answer

This code accepts the variables `a` and `b`, and if `b` is positive, it adds `a` and `b`. However, if `b` is zero or negative, it subtracts them. This decision is made by using the `+` and `-` procedures as the results of an if expression, and then evaluating according to the results of that expression. This is in contrast to a language like Python, which would do something like this:

```
if b > 0: a + b
else: a - b
```

6 Exercise 1.5

6.1 Question

Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

Then he evaluates the expression

```
(test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form if is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

6.2 Answer

In either type of language, `(define (p) (p))` is an infinite loop. However, a normal-order language will encounter the special form, return `0`, and never evaluate `(p)`. An applicative-order language evaluates the arguments to `(test 0 (p))`, thus triggering the infinite loop.

7 Exercise 1.6

7.1 Text code

```
(define (abs x)
  (if (< x 0)
^^I  (- x)
^^I  x))
```

```
(define (average x y)
  (/ (+ x y) 2))
```

```
<<average>>
(define (improve guess x)
  (average guess (/ x guess)))

<<square>>
<<abs>>
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (sqrt x)
  (sqrt-iter 1.0 x))
```

7.2 Question

Exercise 1.6: Alyssa P. Hacker doesn't see why if needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of cond?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of if:

```
(define (new-if predicate
                then-clause
                else-clause)
```

```
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
;; => 5

(new-if (= 1 1) 0 5)
;; => 0
```

Delighted, Alyssa uses new-if to rewrite the square-root program:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x) x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

7.3 Answer

Using Alyssa's `new-if` leads to an infinite loop because the recursive call to `sqrt-iter` is evaluated before the actual call to `new-if`. This is because `if` and `cond` are special forms that change the way evaluation is handled; whichever branch is chosen leaves the other branches unevaluated.

8 Exercise 1.7

8.1 Text

```
(define (mean-square x y)
  (average (square x) (square y)))
```

8.2 Question

The good-enough? test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing good-enough? is to watch how guess changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

8.3 Diary

8.3.1 Solving

My original answer was this, which compares the previous iteration until the new and old are within an arbitrary dx.

```
<<txt-sqrt>>
(define (inferior-good-enough? guess lastguess)
  (<=
   (abs (-
         (/ lastguess guess)
         1))
   0.0000000000001)) ; dx
(define (new-sqrt-iter guess x lastguess) ;; Memory of previous value
  (if (inferior-good-enough? guess lastguess)
      guess
      (new-sqrt-iter (improve guess x) x guess)))
(define (new-sqrt x)
  (new-sqrt-iter 1.0 x 0))
```

This solution can correctly find small and large numbers:

```
<<inferior-good-enough>>
(new-sqrt 10000000000000)
```

```
<<try-these>>
<<inferior-good-enough>>
(try-these new-sqrt '(0.01 0.0001 0.000001 0.00000001 0.0000000001))
```

However, I found this solution online that isn't just simpler but automatically reaches the precision limit of the system:

```
<<txt-sqrt>>
(define (best-good-enough? guess x)
   (= (improve guess x) guess))
```

8.3.2 Imroving (sqrt) by avoiding extra (improve) call

1. Non-optimized

```
(use-modules (ice-9 format))
(load "../mattbench.scm")
(define (average x y)
  (/ (+ x y) 2))
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? guess x)
   (= (improve guess x) guess)) ;; improve call 1
```

```scheme
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x))) ;; call 2
(define (sqrt x)
  (sqrt-iter 1.0 x))
(newline)
(display (mattbench (λ() (sqrt 69420)) 400000000))
(newline)
;; 4731.30 <- Benchmark results
```

2. Optimized

```scheme
(use-modules (ice-9 format))
(load "../mattbench.scm")
(define (average x y)
  (/ (+ x y) 2))
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? guess nextguess x)
  (= nextguess guess))
(define (sqrt-iter guess x)
  (let ((nextguess (improve guess x)))
    (if (good-enough? guess nextguess x)
        guess
        (sqrt-iter nextguess x))))
(define (sqrt x)
  (sqrt-iter 1.0 x))
(newline)
(display (mattbench (λ() (sqrt 69420)) 400000000))
(newline)
```

3. Benchmark results

Unoptimized	4731.30
Optimized	2518.44

8.4 Answer

The current method has decreasing accuracy with smaller numbers. Notice the steady divergence from correct answers here (should be decreasing powers of 0.1):

```scheme
<<txt-sqrt>>
<<try-these>>
(try-these sqrt 0.01 0.0001 0.000001 0.00000001 0.0000000001)
```

And for larger numbers, an infinite loop will eventually be reached. 10^{12} can resolve, but 10^{13} cannot.

```
<<txt-sqrt>>
(sqrt 1000000000000)
```

So, my definition of `sqrt`:

```
<<average>>
(define (improve guess x)
  (average guess (/ x guess)))
(define (good-enough? guess x)
   (= (improve guess x) guess))
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

```
<<try-these>>
<<sqrt>>
(try-these sqrt '(0.01 0.0001 0.000001 0.00000001 0.0000000001))
```

9 Exercise 1.8

9.1 Question

Newton's method for cube roots is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value:

$$\frac{\frac{x}{y^2} + 2y}{3} \tag{1}$$

Use this formula to implement a cube-root procedure analogous to the square-root procedure. (In 1.3.4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

9.2 Diary

My first attempt works, but needs an arbitrary limit to stop infinite loops:

```
<<square>>
<<try-these>>
(define (cb-good-enough? guess x)
  (= (cb-improve guess x) guess))
(define (cb-improve guess x)
  (/
   (+
    (/ x (square guess))
```

```
    (* guess 2))
   3))
(define (cbrt-iter guess x counter)
  (if (or (cb-good-enough? guess x) (> counter 100))
      guess
      (begin
        (cbrt-iter (cb-improve guess x) x (+ 1 counter)))))
(define (cbrt x)
  (cbrt-iter 1.0 x 0))

(try-these cbrt 7 32 56 100)
```

However, this will hang on an infinite loop when trying to run `(cbrt 100)`. I speculate it's a floating point precision issue with the "improve" algorithm. So to avoid it I'll just keep track of the last guess and stop improving when there's no more change occurring. Also while researching I discovered that (again due to floating point) `(cbrt -2)` loops forever unless you initialize your guess with a slightly different value, so let's do 1.1 instead.

9.3 Answer

```
<<square>>
(define (cb-good-enough? nextguess guess lastguess x)
  (or (= nextguess guess)
      (= nextguess lastguess)))
(define (cb-improve guess x)
  (/
   (+
    (/ x (square guess))
    (* guess 2))
   3))
(define (cbrt-iter guess lastguess x)
  (define nextguess (cb-improve guess x))
  (if (cb-good-enough? nextguess guess lastguess x)
      nextguess
      (cbrt-iter nextguess guess x)))
(define (cbrt x)
  (cbrt-iter 1.1 9999 x))
```

```
<<cbrt>>
<<try-these>>
(try-these cbrt 7 32 56 100 -2)
```

10 Exercise 1.9

10.1 Question

Each of the following two procedures defines a method for adding two positive integers in terms of the procedures inc, which increments its argument by 1, and dec, which decrements its argument by 1.

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

Using the substitution model, illustrate the process generated by each procedure in evaluating `(+ 4 5)`. Are these processes iterative or recursive?

10.2 Answer

The first procedure is recursive, while the second is iterative though tail-recursion.

10.2.1 recursive procedure

```
(+ 4 5)
(inc (+ 3 5))
(inc (inc (+ 2 5)))
(inc (inc (inc (+ 1 5))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
9
```

10.2.2 iterative procedure

```
(+ 4 5)
(+ 3 6)
(+ 2 7)
(+ 1 8)
(+ 0 9)
9
```

11 Exercise 1.10

11.1 Question

The following procedure computes a mathematical function called Ackermann's function.

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

What are the values of the following expressions?

```
(A 1 10)
(A 2 4)
(A 3 3)
```

```
<<ackermann>>
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

Give concise mathematical definitions for the functions computed by the procedures `f`, `g`, and `h` for positive integer values of n. For example, `(k n)` computes $5n^2$.

11.2 Answer

11.2.1 `f`

```
<<try-these>>
<<EX1-10-defs>>
(try-these f 1 2 3 10 15 20)
```

$$f(n) = 2n$$

11.2.2 `g`

```
<<try-these>>
<<EX1-10-defs>>
(try-these g 1 2 3 4 5 6 7 8)
```

$$g(n) = 2^n$$

11.2.3 `h`

```
<<try-these>>
<<EX1-10-defs>>
(try-these h 1 2 3 4)
```

It took a while to figure this one out, just because I didn't know the term. This is repeated exponentiation. This operation is to exponentiation, what exponentiation is to multiplication. It's called either *tetration* or *hyper-4* and has no formal notation, but two common ways would be these:

$$h(n) = 2 \uparrow\uparrow n$$

$$h(n) = {}^{n}2$$

12 Exercise 1.11

12.1 Question

A function f is defined by the rule that:

$$f(n) = n \text{ if } n < 3$$

and

$$f(n) = f(n-1) + 2f(n-2) + 3f(n-3) \text{ if } n \geq 3$$

Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

12.2 Answer

12.2.1 Recursive

```
(define (fr n)
  (if (< n 3)
      n
      (+      (fr (- n 1))
         (* 2 (fr (- n 2)))
         (* 3 (fr (- n 3))))))
```

```
<<try-these>>
<<EX1-11-fr>>
(try-these fr 1 3 5 10)
```

12.2.2 Iterative

1. Attempt 1

```scheme
;; This seems like it could be better
(define (fi n)
  (define (formula l)
    (let ((a (car l))
          (b (cadr l))
          (c (caddr l)))
      (+ a
         (* 2 b)
         (* 3 c))))
  (define (iter l i)
    (if (= i n)
        (car l)
        (iter (cons (formula l) l)
              (+ 1 i))))
  (if (< n 3)
      n
      (iter '(2 1 0) 2)))
```

```scheme
<<try-these>>
<<EX1-11-fi>>
(try-these fi 1 3 5 10)
```

It works but it seems wasteful.

2. Attempt 2

```scheme
(define (fi2 n)
  (define (formula a b c)
      (+ a
         (* 2 b)
         (* 3 c)))
  (define (iter a b c i)
    (if (= i n)
        a
        (iter (formula a b c)
              a
              b
              (+ 1 i))))
  (if (< n 3)
      n
      (iter 2 1 0 2)))
```

```scheme
<<try-these>>
<<EX1-11-fi2>>
(try-these fi2 1 3 5 10)
```

I like that better.

13 Exercise 1.12

13.1 Question

The following pattern of numbers is called Pascal's triangle.

Pretend there's a Pascal's triangle here.

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

13.2 Answer

I guess I'll rotate the triangle 45 degrees to make it the top-left corner of an infinite spreadsheet.

```
(define (pascal x y)
  (if (or (= x 0)
          (= y 0))
      1
      (+ (pascal (- x 1) y)
         (pascal x (- y 1)))))
```

```
<<try-these>>
<<pascal-rec>>
(let ((l (iota 8)))
  (map (λ (row)
         (map (λ (xy)
                (apply pascal xy))
              row))
       (map (λ (x)
              (map (λ (y)
                     (list x y))
                   l))
            l)))
```

The test code was much harder to write than the actual solution.

14 Exercise 1.13

14.1 Question

Prove that Fib(n) is the closest integer to $\frac{n}{\sqrt{5}}$ where Phi is $\frac{1+\sqrt{5}}{2}$. Hint: let $= \frac{1-\sqrt{5}}{2}$. Use induction and the definition of the Fibonacci numbers to prove that

$$\text{Fib}(n) = \frac{{}^{n} - {}^{n}}{\sqrt{5}}$$

14.2 Answer

I don't know how to write a proof yet, but I can make functions to demonstrate it.

14.2.1 Fibonacci number generator

```
(define (fib-iter n)
  (define (iter i a b)
    (if (= i n)
        b
    (iter (+ i 1)
          b
          (+ a b))))
  (if (<= n 2)
      1
      (iter 2 1 1)))
```

14.2.2 Various algorithms relating to the question

```
<<sqrt>>
(define sqrt5
  (sqrt 5))
(define phi
  (/ (+ 1 sqrt5) 2))
(define upsilon
  (/ (- 1 sqrt5) 2))
(define (fib-phi n)
  (/ (- (expt phi n)
        (expt upsilon n))
     sqrt5))
```

```
(use-srfis '(1))
<<fib-iter>>
<<fib-phi>>
<<try-these>>

(let* ((vals (drop (iota 21) 10))
       (fibs (map fib-iter vals))
       (approx (map fib-phi vals)))
  (zip vals fibs approx))
```

You can see they follow closely. Graphing the differences, it's just an exponential curve at very low values, presumably following the exponential increase of the Fibonacci sequence itself.

15 Exercise 1.14

Below is the default version of the count-change function. I'll be aggressively modifying it in order to get a graph out of it.

```
(define (count-change amount)
  (cc amount 5))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0)
             (= kinds-of-coins 0))
         0)
        (else
         (+ (cc amount (- kinds-of-coins 1))
            (cc (- amount (first-denomination
                           kinds-of-coins))
                kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

15.1 Question

Draw the tree illustrating the process generated by the count-change procedure of 1.2.2 in making change for 11 cents.

15.2 Answer

I want to generate this graph algorithmically.

```
;; cursed global
(define bubblecounter 0)
;; Returns # of ways change can be made
;; "Helper" for (cc)
(define (count-change amount)
  (display "digraph {\n") ;; start graph
  (cc amount 5 0)
```

```
  (display "}\n") ;; end graph
  (set! bubblecounter 0))

;; GraphViz output
;; Derivative: https://stackoverflow.com/a/14806144
(define (cc amount kinds-of-coins oldbubble)
  (let ((recur (lambda (new-amount new-kinds)
                 (begin
                   (display "\"") ;; Source bubble
                   (display `(,oldbubble ,amount ,kinds-of-coins))
                   (display "\"")
                   (display " -> ") ;; arrow pointing from parent to child
                   (display "\"") ;; child bubble
                   (display `(,bubblecounter ,new-amount ,new-kinds))
                   (display "\"")
                   (display "\n")
                   (cc new-amount new-kinds bubblecounter)))))
    (set! bubblecounter (+ bubblecounter 1))
    (cond ((= amount 0) 1)
          ((or (< amount 0) (= kinds-of-coins 0)) 0)
          (else (+
                 (recur amount (- kinds-of-coins 1))
                 (recur (- amount
                           (first-denomination kinds-of-coins))
                        kinds-of-coins))))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

I'm not going to include the full printout of the `(count-change 11)`, here's an example of what this looks like via `1`.

```
<<count-change-graphviz>>
(count-change 1)
```

```
digraph {
"(0 1 5)" -> "(1 1 4)"
"(1 1 4)" -> "(2 1 3)"
"(2 1 3)" -> "(3 1 2)"
"(3 1 2)" -> "(4 1 1)"
"(4 1 1)" -> "(5 1 0)"
"(4 1 1)" -> "(6 0 1)"
```

```
"(3 1 2)" -> "(7 -4 2)"
"(2 1 3)" -> "(8 -9 3)"
"(1 1 4)" -> "(9 -24 4)"
"(0 1 5)" -> "(10 -49 5)"
}
```

So, the graph of `(count-change 11)` is:

15.3 Question 2

What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

15.4 Answer 2

Let's look at this via the number of function calls needed for value `n`. Instead of returning an integer, I'll return a pair where `car` is the number of ways to count change, and `cdr` is the number of function calls that have occurred down that branch of the tree.

```
(define (count-calls amount)
  (cc-calls amount 5))

(define (cc-calls amount kinds-of-coins)
  (cond ((= amount 0) '(1 . 1))
        ((or (< amount 0)
             (= kinds-of-coins 0))
         '(0 . 1))
        (else
         (let ((a (cc-calls amount (- kinds-of-coins 1)))
               (b (cc-calls (- amount (first-denomination
                                       kinds-of-coins))
                            kinds-of-coins)))
           (cons (+ (car a)
                    (car b))
                 (+ 1
                    (cdr a)
                    (cdr b)))))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

```
(use-srfis '(1))
<<cc-calls>>
(let* ((vals (drop (iota 101) 1))
       (mine (map count-calls vals)))
  (zip vals (map car mine) (map cdr mine)))
```

I believe the space to be $\Theta(n + d)$ as the function calls count down the denominations before counting down the change. However I notice most answers describe $\Theta(n)$ instead, maybe I'm being overly pedantic and getting the wrong answer.

My issues came finding the time. The book describes the meaning and properties of Θ notation in Section 1.2.3. However, my lack of formal math education made realizing the significance of this passage difficult. For one, I didn't understand that $k_1 f(n) \leq R(n) \leq k_2 f(n)$ means "you can find the Θ by proving that a graph of the algorithm's resource usage is bounded by two identical functions multiplied by constants." So, the graph of resource usage for an algorithm with $\Theta(n^2)$ will by bounded by lines of $n^2 \times someconstant$, the top boundary's constant being larger than the small boundary. These are arbitrarily chosen constants, you're just proving that the function behaves the way you think it does.

Overall, finding the Θ and Ω and O notations (they are all different btw!) is about aggressively simplifying to make a very general statement about the behavior of the algorithm.

I could tell that a "correct" way to find the Θ would be to make a formula which describes the algorithm's function calls for given input and denominations. This is one of the biggest time sinks, although I had a lot of fun and learned a lot. In the end, with some help from Jach in a Lisp Discord, I had the following formula:

$$\sum_{i=1}^{ceil(n/val(d))} T(n - val(d) * i, d)$$

But I wasn't sure where to go from here. The graphs let me see some interesting trends, though I didn't get any closer to an answer in the process.

By reading on other websites, I knew that you could find Θ by obtaining a formula for $R(n)$ and removing constants to end up with a term of interest. For example, if your algorithm's resource usage is $\frac{n^2+7n}{5}$, this demonstrates $\Theta(n^2)$. So I know a formula **without** a $\sum$ would give me the answer I wanted. It didn't occur to me that it might be possible to use calculus to remove the $\sum$ from the equation. At this point I knew I was stuck and decided to look up a guide.

After seeing a few solutions that I found somewhat confusing, I landed on this awesome article from Codology.net. They show how you can remove the summation, and proposed this equation for count-change with 5 denominations:

$$T(n,5) = \frac{n}{50} + 1 + \sum_{i=0}^{n/50} T(n - 50i, 1)$$

Which, when expanded and simplified, demonstrates $\Theta(n^5)$ for 5 denominations.

Overall I'm relieved that I wasn't entirely off, given I haven't done math work like this since college. It's inspired me to restart my remedial math courses, I don't think I really grasped the nature of math as a tool of empowerment until now.

16 Exercise 1.15

16.1 Question 1

The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x$ x if x is sufficiently small, and the trigonometric identity $\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$ to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0)))))
```

How many times is the procedure `p` applied when `(sine 12.15)` is evaluated?

16.2 Answer 1

Let's find out!

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      (cons angle 0)
      (let ((x (sine (/ angle 3.0))))
        (cons (p (car x)) (+ 1 (cdr x))))))
```

```
<<1-15-p-measure>>
(let ((xy (sine 12.15)))
  (list (car xy) (cdr xy)))
```

`p` is evaluated 5 times.

16.3 Question 2

What is the order of growth in space and number of steps (as a function of `a`) used by the process generated by the sine procedure when `(sine a)` is evaluated?

16.4 Answer 2

```
(use-srfis '(1))
<<1-15-p-measure>>
(let* ((vals (iota 300 0.1 0.1))
       (sines (map (λ (i)
                     (cdr (sine i)))
                   vals)))
  (zip vals sines))
```

#+end$_{src}$

```
(use-srfis '(1))
<<1-15-p-measure>>
(let* ((vals (iota 10 0.1 0.1))
       (sines (map (λ (i)
                     (cdr (sine i)))
                   vals)))
  (zip vals sines))
```

Example output:

```
reset # helps with various issues in execution
set xlabel 'values of x'
set logscale x
set key top left
set style fill solid 1.00 border
set style function fillsteps below

f(x) = log(x) + 2.3

plot data using 1:2 with fillsteps title 'function calls', \
     data using 1:(f($1)) with lines title 'log(x) + 2. 3'
```

This graph shows that the number of times `sine` will be called is logarithmic.

- 0.1 to 0.2 are divided once
- 0.3 to 0.8 are divided twice
- 0.9 to 2.6 are divided three times
- 2.7 to 8 are divided four times
- 8.5 to 23.8 are divided five times

Given that the calls to `p` get stacked recursively, like this:

```
(sine 12.15)
(p (sine 4.05))
(p (p (sine 1.35)))
(p (p (p (sine 0.45))))
(p (p (p (p (sine 0.15)))))
(p (p (p (p (p (sine 0.05))))))
(p (p (p (p (p 0.05)))))
(p (p (p (p 0.14950000000000002))))
(p (p (p 0.43513455050000005)))
(p (p 0.9758465331678772))
(p -0.7895631144708228)
-0.39980345741334
```

So I argue the space and time is $\Theta(\log(n))$

We can also prove this for the time by benchmarking the function:

```
;; This execution takes too long for org-mode, so I'm doing it
;; externally and importing the results
(use-srfis '(1))
(use-modules (ice-9 format))
(load "../../mattbench.scm")
<<1-15-deps>>
(let* ((vals (iota 300 0.1 0.1))
       (times (map (λ (i)
                      (mattbench (λ () (sine i)) 1000000))
                    vals)))
  (with-output-to-file "sine-bench.dat" (λ ()
     (map (λ (x y)
            (format #t "~s~/~s~%" x y))
          vals times))))
```

```
reset # helps with various issues in execution
set xtics 0.5
set xlabel 'values of x'
set logscale x
set key top left
set style fill solid 1.00 border
#set style function fillsteps below

f(x) = (log(x) * a) + b
fit f(x) 'Ex15/sine-bench.dat' using 1:2 via a,b

plot 'Ex15/sine-bench.dat' using 1:2 with fillsteps title
↪  'time to execute', \
     'Ex15/sine-bench.dat' using 1:(f($1)) with lines title
↪  sprintf('(log(x) * %.2f) + %.2f', a, b)
```

17 Exercise 1.16

17.1 Text

```scheme
(define (expt-rec b n)
  (if (= n 0)
      1
      (* b (expt-rec b (- n 1)))))

(define (expt-iter b n)
  (define (iter counter product)
    (if (= counter 0)
        product
        (iter (- counter 1)
              (* b product))))
  (iter n 1))

(define (fast-expt b n)
  (cond ((= n 0)
         1)
        ((even? n)
         (square (fast-expt b (/ n 2))))
        (else
         (* b (fast-expt b (- n 1))))))
```

17.2 Question

Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does fast-expt. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a , and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

17.3 Diary

First I made this program which tries to use a false equivalence:

$$ab^2 = (a+1)b^{n-1}$$

```scheme
<<square>>
(define (fast-expt-iter b n)
  (define (iter b n a)
    (format #t "~&|~s~/~/|~s~/~/|~s|~%" b n a)
```

```
    (cond ((= n 1) (begin (format #t "~&|~s~/~/|~s~/~/|~s|~%" (* b a) 1 1)
                          (* b a)))
          ((even? n) (iter (square b)
                           (/ n 2)
                           a))
          (else (iter b (- n 1) (+ a 1)))))
  (format #t "|~a~/|~a~/|~a|~%" "base" "power" "variable")
  (format #t "~&|--|--|--|~%")
  (iter b n 1))
```

```
<<fast-expt-iter-fail1>>
<<try-these>>
(fast-expt-iter 2 6)
```

Here's what the internal state looks like during 2^6 (correct answer is 64):

17.4 Answer

There are two key transforms to a faster algorithm. The first was already shown in the text:

$$ab^n \to a(b^2)^{n/2}$$

The second which I needed to deduce was this:

$$ab^n \to ((a \times b) \times b)^{n-1}$$

The solution essentially follows this logic:

- initialize a to 1
- If n is 1, return $b * a$
- else if n is even, halve n, square b, and iterate
- else n is odd, so subtract 1 from n and $a \to a \times b$

```
<<square>>
(define (fast-expt-iter b n)
  (define (iter b n a)
    (cond ((= n 1) (* b a))
          ((even? n) (iter (square b)
                           (/ n 2)
                           a))
          (else (iter b (- n 1) (* b a)))))
  (iter b n 1))
```

```
<<fast-expt-iter>>
<<try-these>>
(try-these (λ(x) (fast-expt-iter 3 x)) (cdr (iota 11)))
```

18 Exercise 1.17

18.1 Question

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the expt procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes a number of steps that is linear in b. Now suppose we include, together with addition, operations double, which doubles an integer, and halve, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to fast-expt that uses a logarithmic number of steps.

18.2 Answer

```
(define (double x)
  (+ x x))
(define (halve x)
  (/ x 2))
(define (fast-mult-rec a b)
  (cond ((= b 0) 0)
        ((even? b)
         (double (fast-mult-rec a (halve b)))) ; This was kind of a stretch to think of.G
         ;(fast-mult (double a) (halve b))) <== My first instinct is iterative
        (else (+ a (fast-mult-rec a (- b 1))))))
```

Proof it works:

```
<<fast-mult-rec>>
<<try-these>>
(try-these (λ(x) (fast-mult-rec 3 x)) (cdr (iota 11)))
```

19 Exercise 1.18

19.1 Question

Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

19.2 Diary

19.2.1 Comparison benchmarks:

```
(load "../mattbench.scm")
<<fast-mult-iter>>
<<fast-mult-rec>>
<<print-table>>
(print-table (list (list "fast-mult-rec" "fast-mult-iter")
                   (list (mattbench (λ() (fast-mult-rec 32 32)) 10000000)
                         (mattbench (λ() (fast-mult 32 32)) 10000000)))
             #:colnames #t)
```

"fast-mult-rec"	"fast-mult-iter"
196.89	166.35

So the iterative version takes 0.84 times less to do 32×32.

19.2.2 Hall of shame

Some of my *very* incorrect ideas:

$$ab = (a+1)(b-1)$$

$$ab = \big(a + \left(\frac{a}{2}\right)(b-1)\big)$$

$$ab + c = \big(a(b-1) + (b+c)\big)$$

19.3 Answer

```
(define (double x)
  (+ x x))
(define (halve x)
  (/ x 2))
(define (fast-mult a b)
  (define (iter a b c)
    (cond ((= b 0) 0)
          ((= b 1) (+ a c))
          ((even? b)
           (iter (double a) (halve b) c))
          (else (iter a (- b 1) (+ a c)))))
  (iter a b 0))
```

```
<<fast-mult-iter>>
<<try-these>>
(try-these (λ(x) (fast-mult 3 x)) (cdr (iota 11)))
```

20 Exercise 1.19

20.1 Question

There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state variables a and b in the `fib-iter` process of section 1-2-2:

$a < -a + b$ and $b < -a$

Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair _Fib$_{(\text{n + 1})}$ and _Fib$_{(\text{n})}$. In other words, the Fibonacci numbers are produced by applying T^n, the nth power of the transformation T, starting with the pair (1,0). Now consider T to be the special case of p = 0 and q = 1 in a family of transformations $T_{(pq)}$, where $T_{(pq)}$ transforms the pair (a,b) according to $a < -bq + aq + ap$ and $b < -bp + aq$. Show that if we apply such a transformation $T_{(pq)}$ twice, the effect is the same as using a single transformation $T_{(p'q')}$ of the same form, and compute p' and q' in terms of p and q. This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring, as in the 'fast-expt' procedure. Put this all together to complete the following procedure, which runs in a logarithmic number of steps:

```scheme
(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   <??>      ; compute p'
                   <??>      ; compute q'
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

20.2 Diary

More succinctly put:

$$\text{Fib}_n \begin{cases} a \leftarrow a + b \\ b \leftarrow a \end{cases}$$

$$\text{Fib-iter}_{abpq}\begin{cases} a \leftarrow bq + aq + ap \\ b \leftarrow bp + aq \end{cases}$$

`(T)` returns a transformation function based on the two numbers in the attached list. so `(T 0 1)` returns a fib function.

```
(define (T p q)
  (λ (a b)
    (cons (+ (* b q) (* a q) (* a p))
          (+ (* b p) (* a q)))))

(define T-fib
  (T 0 1))

;; Repeatedly apply T functions:
(define (Tr f n)
  (Tr-iter f n 0 1))
(define (Tr-iter f n a b)
  (if (= n 0)
      a
      (let ((l (f a b)))
        (Tr-iter f (- n 1) (car l) (cdr l)))))
```

$$\text{T}_{pq} : a, b \mapsto \begin{cases} a \leftarrow bq + aq + ap \\ b \leftarrow bp + aq \end{cases}$$

```
<<T-func>>
<<try-these>>
(try-these (λ (x) (Tr (T 0 1) x)) (cdr (iota 11)))
```

20.3 Answer

```
(define (fib-rec n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib-rec (- n 1))
                 (fib-rec (- n 2))))))
(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
```

```
                    (+ (* p p)
                       (* q q))      ; compute p'
                    (+ (* p q)
                       (* q q)
                       (* q p))      ; compute q'
                    (/ count 2)))
         (else (fib-iter (+ (* b q) (* a q) (* a p))
                         (+ (* b p) (* a q))
                         p
                         q
                         (- count 1)))))
```

21 Exercise 1.20

21.1 Text

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

21.2 Question

The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative `gcd` procedure given above. Suppose we were to interpret this procedure using normal-order evaluation, as discussed in 1.1.5. (The normal-order-evaluation rule for `if` is described in Exercise 1.5.) Using the substitution method (for normal order), illustrate the process generated in evaluating `(gcd 206 40)` and indicate the remainder operations that are actually performed. How many remainder operations are actually performed in the normal-order evaluation of `(gcd 206 40)`? In the applicative-order evaluation?

21.3 Answer

I struggled to understand this, but the key here is that normal-order evaluation causes the unevaluated expressions to be duplicated, meaning they get evaluated multiple times.

21.3.1 Applicative order

```
call (gcd 206 40)
(if)
(gcd 40 (remainder 206 40))
eval remainder before call
call (gcd 40 6)
```

```
(if)
(gcd 6 (remainder 40 6))
eval remainder before call
call (gcd 6 4)
(if)
(gcd 2 (remainder 4 2))
eval remainder before call
call (gcd 2 0)
(if)
;; => 2
```

```
;; call gcd
(gcd 206 40)

;; eval conditional
(if (= 40 0)
    206
    (gcd 40 (remainder 206 40)))

;; recurse
(gcd 40 (remainder 206 40))

; encounter conditional
(if (= (remainder 206 40) 0)
    40
    (gcd (remainder 206 40)
         (remainder 40 (remainder 206 40))))

; evaluate 1 remainder
(if (= 6 0)
    40
    (gcd (remainder 206 40)
         (remainder 40 (remainder 206 40))))

; recurse
(gcd (remainder 206 40)
     (remainder 40 (remainder 206 40)))

; encounter conditional
(if (= (remainder 40 (remainder 206 40)) 0)
    (remainder 206 40)
    (gcd (remainder 40 (remainder 206 40))
         (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))

; eval 2 remainder
```

```
(if (= 4 0)
    (remainder 206 40)
    (gcd (remainder 40 (remainder 206 40))
         (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))

; recurse
(gcd (remainder 40 (remainder 206 40))
     (remainder (remainder 206 40) (remainder 40 (remainder 206 40))))

; encounter conditional
(if (= (remainder (remainder 206 40) (remainder 40 (remainder 206 40))) 0)
    (remainder 40 (remainder 206 40))
    (gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
         (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40))))))

; eval 4 remainders
(if (= 2 0)
    (remainder 40 (remainder 206 40))
    (gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
         (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40))))))

; recurse
(gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
     (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))

; encounter conditional
(if (= (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))) 0)
    (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
    (gcd (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))) (remainder a  (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))))

; eval 7 remainders
(if (= 0 0)
    (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
```

```
    (gcd (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))) (remainder a  (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))))))

; eval 4 remainders
(remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
; => 2
```

So, in normal-order eval, remainder is called 18 times, while in applicative order it's called 5 times.

22 Exercise 1.21

22.1 Text

```
<<square>>
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n)
         n)
        ((divides? test-divisor n)
         test-divisor)
        (else (find-divisor
               n
               (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))
```

22.2 Question

Use the smallest-divisor procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

```
<<find-divisor-txt>>
(map smallest-divisor '(199 1999 19999))
```

23 Exercise 1.22

23.1 Question

Most Lisp implementations include a primitive called runtime that returns an integer that specifies the amount of time the system has been running (measured, for example, in microseconds). The following timed-prime-test procedure, when called with an integer n, prints n and checks to see if n is prime. If n is prime, the procedure prints three asterisks followed by the amount of time used in performing the test.

```
<<find-divisor-txt>>
(define (prime? n)
  (= n (smallest-divisor n)))
```

```
<<prime-smallest-divisor>>
(define (timed-prime-test n)
  (newline)
  (display n) ;; Guile compatible \downarrow
  (start-prime-test n (get-internal-run-time)))
(define (start-prime-test n start-time)
  (if (prime? n)
      (begin
        (report-prime (- (get-internal-run-time)
                         start-time))
        n)
      #f))
(define (report-prime elapsed-time)
  (display ” *** ”)
  (display elapsed-time))
```

Using this procedure, write a procedure search-for-primes that checks the primality of consecutive odd integers in a specified range. Use your procedure to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth of $\Theta(\sqrt{n})$, you should expect that testing for primes around 10,000 should take about $\sqrt{10}$ times as long as testing for primes around 1000. Do your timing data bear this out? How well do the data for 100,000 and 1,000,000 support the $\Theta(\sqrt{n})$ prediction? Is your result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation?

23.2 Answer

23.2.1 Part 1

So this question is a little funky, because modern machines are so fast that the single-run times can seriously vary.

```
<<timed-prime-test-txt>>
(define (search-for-primes minimum goal)
  (define m (if (even? minimum)
                (+ minimum 1)
```

```
                (minimum)))
  (search-for-primes-iter m '() goal))
(define (search-for-primes-iter n lst goal)
  (if (= goal 0)
      lst
      (let ((x (timed-prime-test n)))
        (if (not (equal? x #f))
            (search-for-primes-iter (+ n 2) (cons x lst) (- goal 1))
            (search-for-primes-iter (+ n 2) lst goal)))))
```

```
<<search-primes-basic>>
(let ((lt1000-1 (search-for-primes 1000 3)))
  (list "Primes > 1000" lt1000-1))
```

```
1001
1003
1005
1007
1009 *** 1651
1011
1013 *** 1425
1015
1017
1019 *** 1375
```

There's proof it works. And here are the answers to the question:

```
<<search-primes-basic>>
(let ((lt1000-1 (search-for-primes 1000 3))
      (lt10000-1 (search-for-primes 10000 3))
      (lt100000-1 (search-for-primes 100000 3))
      (lt100000000-1 (search-for-primes 1000000 3)))
  (list
   (list "Primes > 1000" (reverse lt1000-1))
   (list "Primes > 10000" (reverse lt10000-1))
   (list "Primes > 100000" (reverse lt100000-1))
   (list "Primes > 100000000" (reverse lt100000000-1))
   ))
```

23.2.2 Part 2

Repeatedly re-running, it I see it occasionally jump to twice the time. I'm not happy with this, so I'm going to refactor to use the `mattbench2` utility from the root of the project folder.

```
(define (mattbench2 f n)
  ;; Executes "f" for n times, and returns how long it took.
```

```
  ;; f is a lambda that takes no arguments, a.k.a. a "thunk"

  ;; Returns a dotted pair with car(last execution results) and
↪  cdr(time taken divided by iterations n)

  (define (time-getter) (get-internal-run-time))
  (define start-time (time-getter))
  (define (how-long) (- (time-getter) start-time))

  (define (iter i)
    (f)
    (if (<= i 0)
        (f) ;; return the results of the function
        (iter (- i 1))))

  (list (iter n) ;; result of last call of f
        (/ (how-long) (* n 1.0))));; Divide by iterations so changed
↪  n has no effect
```

I'm going to get some more precise times. First, I need a prime searching variant that doesn't bother benchmarking. This will call `prime?`, which will be bound later since we'll be trying different methods.

```
(define (search-for-primes minimum goal)
  (define m (if (even? minimum)
                (+ minimum 1)
                (minimum)))
  (search-for-primes-iter m '() goal))
(define (search-for-primes-iter n lst goal)
  (if (= goal 0)
      lst
      (let ((x (prime? n)))
        (if (not (equal? x #f))
            (search-for-primes-iter (+ n 2) (cons n lst) (- goal 1))
            (search-for-primes-iter (+ n 2) lst goal)))))
```

I can benchmark these functions like so:

```
<<mattbench2>>
<<prime-smallest-divisor>>
<<search-for-primes-untimed>>
<<print-table>>

(define benchmark-iterations 100000)
(define (testit name f)
  (list name
        (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
```

```
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000 3)) benchmark-iterations))))

(print-table
 (list (list "function" ">1000" ">100000" ">1000000" ">10000000")
       (testit "smallest-divisor" search-for-primes))
    #:colnames #t)
```

Here are the results (run externally from Org-Mode):

function	>1000	>100000	>1000000	>10000000
smallest-divisor	5449.59114	20678.07188	53432.40465	122270.00802

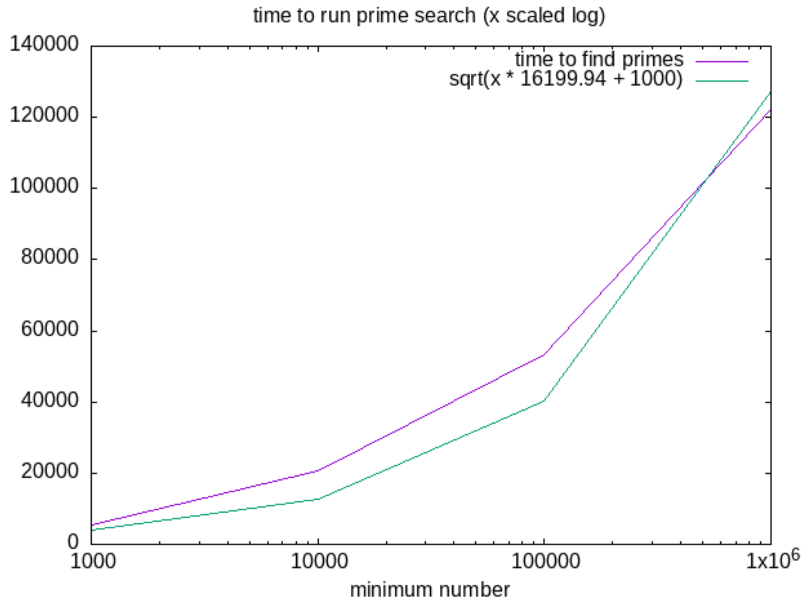
```
;(map (lambda(x y) (list x y))
<<transpose-list>>
(use-srfis '(1))
(zip
     '(1000 10000 100000 1000000)
     (cdar data))
```

1000	5449.59114
10000	20678.07188
100000	53432.40465
1000000	122270.00802

```
reset
set logscale x
set xlabel 'minimum number'
set title 'time to run prime search (x scaled log)'

f(x) = sqrt((x * a) + 1000)
fit f(x) data using 1:2 via a

plot data using 1:2 with lines title 'time to find primes', \
     data using 1:(f($1)) with lines title \
     sprintf('sqrt(x * %.2f + 1000)', a)
```



#+BEGIN_COMMENT

23.2.3 Final comparison

TODO

```
<<mattbench2>>
<<prime-smallest-divisor>>
<<search-for-primes-untimed>>
<<print-table>>

(define benchmark-iterations 1)
(define (testit name f)
  (list name
        (cadr (mattbench2 (λ() (f 1000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 10000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 100000 3)) benchmark-iterations))
        (cadr (mattbench2 (λ() (f 1000000 3)) benchmark-iterations))))

(print-table
 (list (list "function" ">1000" ">100000" ">1000000" ">10000000")
       (testit "smallest-divisor" search-for-primes))
    #:colnames #t)
```

#+END_COMMENT