# SICP Chapter 1

ProducerMatt

# HOW THIS DOCUMENT IS MADE

### TODO

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

(foo 5 3)
```

11

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

### 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 :)

### Current command for conversion

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

## Helpers for org-mode tables

### `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)))
```

`transpose-list`

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

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

`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))))
```

# Exercise 1.1

## Q

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.

### A

```scheme
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
```

## Exercise 1.2

### Q

Translate the following expression into prefix form:

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

### A

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

1/75

## Exercise 1.3

### Text

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

### Q

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

### A

```
<<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))
        ((and (> b a) (>= c a)) (sum-square b c))
        (else "This shouldn't happen")))
```

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

| | |
|---|---|
| (7 5 3) | 74 |
| (7 3 5) | 74 |
| (3 5 7) | 74 |

## Exercise 1.4

### Q

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))
```

### A

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
```

# Exercise 1.5

## Q

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.)

## A

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.

# Exercise 1.6

## Text code

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

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

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

```

```scheme
<<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))
```

## Q

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:

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

Eva demonstrates the program for Alyssa:

```scheme
(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:

```scheme
(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.

## A

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.

## Exercise 1.7

### Text

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

### Q

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?

### A

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)
```

| | |
|---|---|
| 0.01 | 0.10032578510960605 |
| 0.0001 | 0.03230844833048122 |
| 1e-06 | 0.031260655525445276 |
| 1e-08 | 0.03125010656242753 |
| 1e-10 | 0.03125000106562499 |

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

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

1000000.0

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)
```

3162277.6601683795

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

| | |
|---|---|
| 0.01 | 0.1 |
| 0.0001 | 0.01 |
| 1e-06 | 0.001 |
| 1e-08 | 9.999999999999999e-05 |
| 1e-10 | 9.999999999999999e-06 |

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))
```

So, my final 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))
```

| | |
|---|---|
| 0.01 | 0.1 |
| 0.0001 | 0.01 |
| 1e-06 | 0.001 |
| 1e-08 | 9.999999999999999e-05 |
| 1e-10 | 9.999999999999999e-06 |

# Exercise 1.8

## Q

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.)

## A1

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

```
<<square>>
(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)
```

| | |
|---|---|
| 7 | 1.912931182772389 |
| 32 | 3.174802103936399 |
| 56 | 3.825862365544778 |
| 100 | 4.641588833612779 |

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.

## A2

```
<<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)
```

| | |
|---|---|
| 7 | 1.912931182772389 |
| 32 | 3.174802103936399 |

| | |
|---|---|
| 56 | 3.825862365544778 |
| 100 | 4.641588833612779 |
| -2 | -1.2599210498948732 |

# Exercise 1.9

## Q

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?

## A

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

### 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
```

#### iterative procedure

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

## Exercise 1.10

### Q

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)
```

| | |
|---|---|
| (1 10) | 1024 |
| (2 4) | 65536 |
| (3 3) | 65536 |

```
<<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 $5n^2$.

### A

#### f

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

| | |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 10 | 20 |
| 15 | 30 |
| 20 | 40 |

$$f(n) = 2n$$

#### g

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

| | |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |

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

#### h

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

| | |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 16 |

| | |
|---|---|
| 4 | 65536 |

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:

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

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

# Exercise 1.11

## Q

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.

## A

### 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)
```

| | |
|---|---|
| 1 | 1 |
| 3 | 4 |
| 5 | 25 |
| 10 | 1892 |

### 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)))
```

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

| | |
|---|---|
| 1 | 1 |
| 3 | 4 |
| 5 | 25 |
| 10 | 1892 |

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)))
```

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

| | |
|---|---|
| 1 | 1 |
| 3 | 4 |
| 5 | 25 |
| 10 | 1892 |

I like that better.

## Exercise 1.12

### Q

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.

### A

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)))
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 3 | 6 | 10 | 15 | 21 | 28 | 36 |
| 1 | 4 | 10 | 20 | 35 | 56 | 84 | 120 |
| 1 | 5 | 15 | 35 | 70 | 126 | 210 | 330 |
| 1 | 6 | 21 | 56 | 126 | 252 | 462 | 792 |
| 1 | 7 | 28 | 84 | 210 | 462 | 924 | 1716 |
| 1 | 8 | 36 | 120 | 330 | 792 | 1716 | 3432 |

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

## Exercise 1.13

### Q

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

$$\mathrm{Fib}(n) = \frac{\phi^n - \Upsilon^n}{\sqrt{5}}$$

### A

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

#### 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)))
```

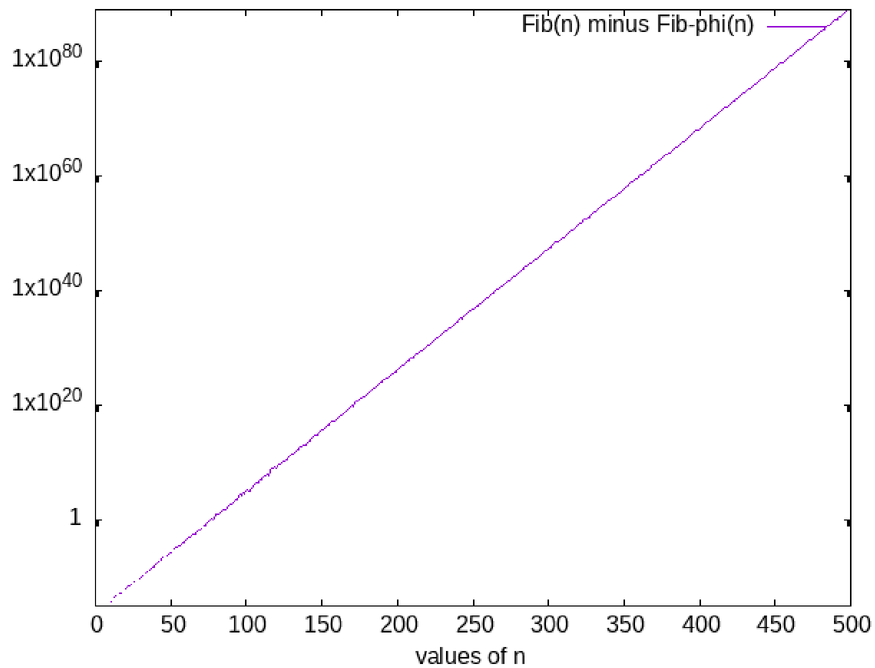
**Various algorithms relating to the question**

```
<<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))
```

| | | |
|---|---|---|
| 10 | 55 | 54.99999999999999 |
| 11 | 89 | 89.0 |
| 12 | 144 | 143.99999999999997 |
| 13 | 233 | 232.99999999999994 |
| 14 | 377 | 377.00000000000006 |
| 15 | 610 | 610.0 |
| 16 | 987 | 986.9999999999998 |
| 17 | 1597 | 1596.9999999999998 |
| 18 | 2584 | 2584.0 |
| 19 | 4181 | 4181.0 |
| 20 | 6765 | 6764.999999999999 |

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.




## 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)
```

```scheme
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

## Q

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

## A

I want to generate this graph algorithmically.

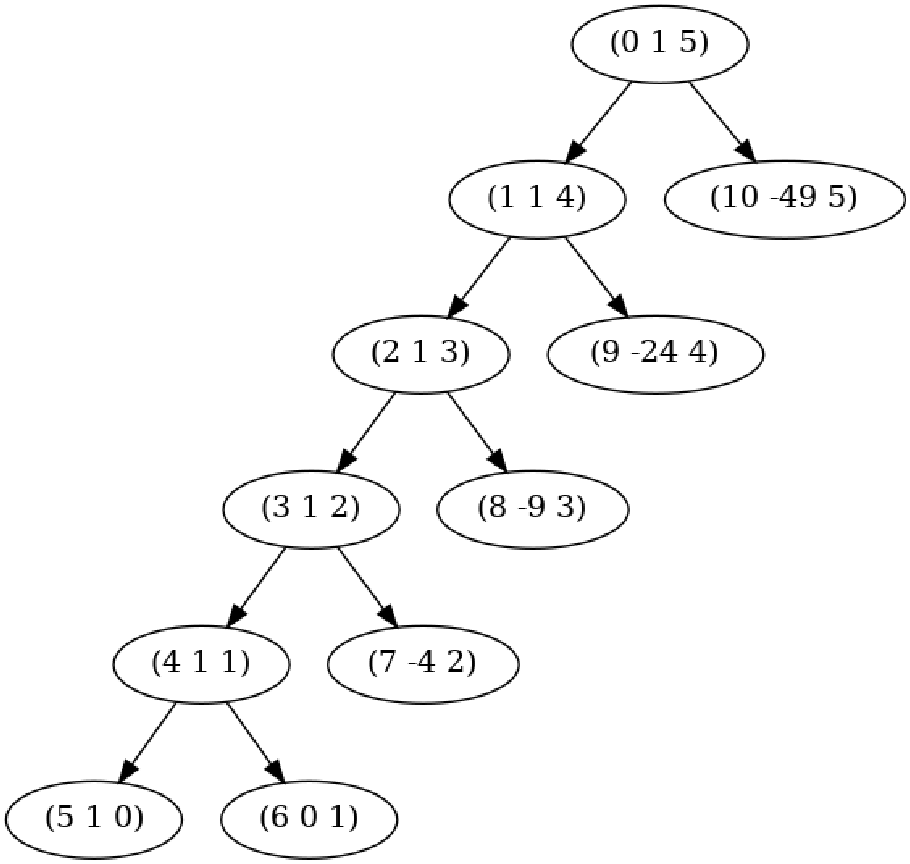
```scheme
;; cursed global
(define bubblecounter 0)
;; Returns # of ways change can be made
;; "Helper" for (cc)
(define (count-change amount)
  (display "digraph {\n") ;; start graph
  (cc amount 5 0)
  (display "}\n") ;; end graph
  (set! bubblecounter 0))

;; 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)))
```

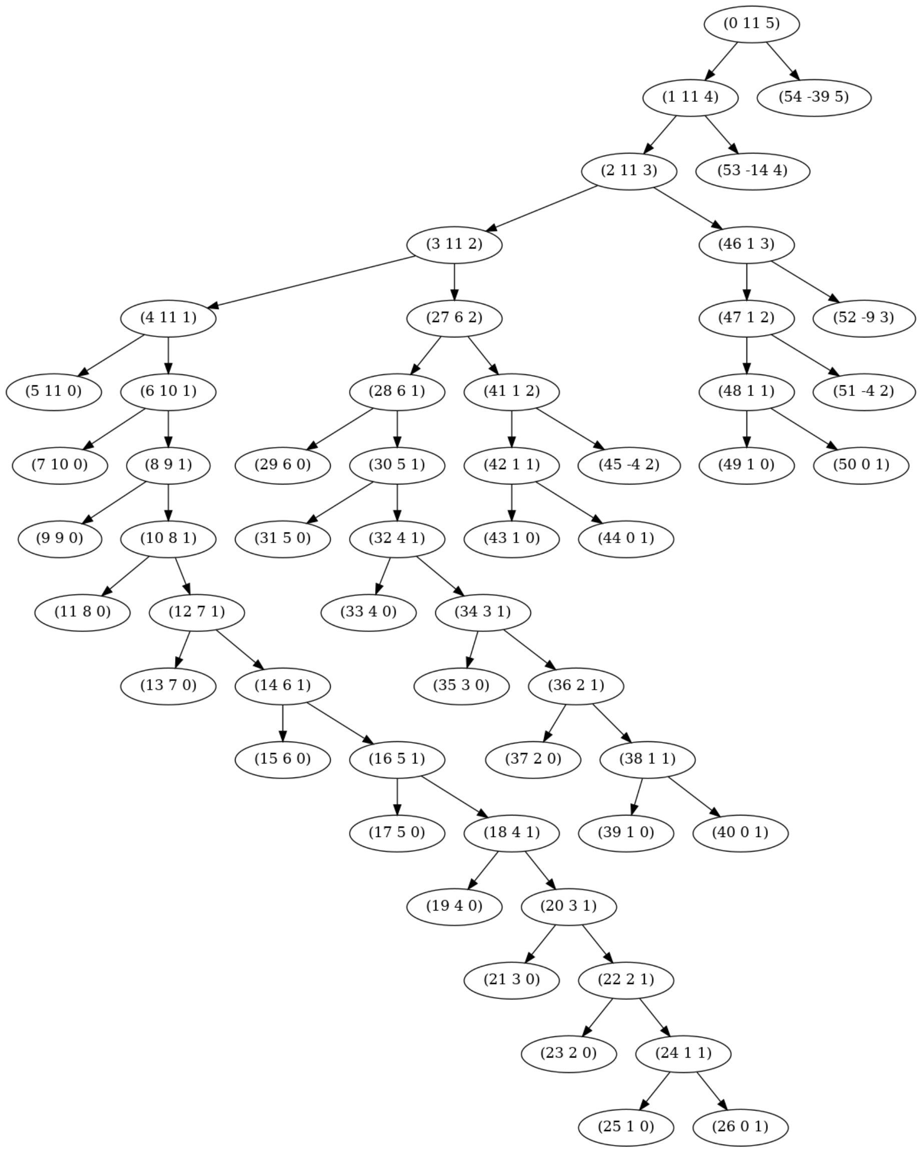
I'm not going to include the full printout of the `(count-change 11)`, here's an example of what this looks like via `1`.

```
<<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:




## Q2

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

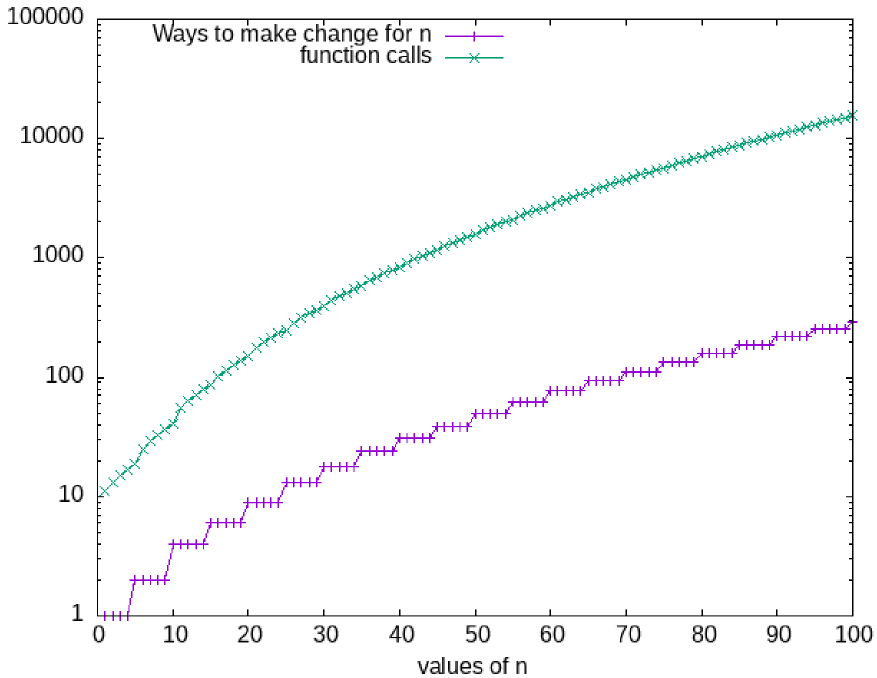
## A2

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)))
```

| | | |
|---|---|---|
| 1 | 1 | (1 . 11) |
| 2 | 1 | (1 . 13) |
| 3 | 1 | (1 . 15) |
| 4 | 1 | (1 . 17) |
| 5 | 2 | (2 . 19) |
| 6 | 2 | (2 . 25) |
| 7 | 2 | (2 . 29) |
| 8 | 2 | (2 . 33) |
| 9 | 2 | (2 . 37) |
| 10 | 4 | (4 . 41) |

```
(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 $\Theta$ 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 $\Theta$ 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 $\Theta$ and $\Omega$ 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 $\Theta$ 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 $\Theta$ 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.

# Exercise 1.15

## Q1

The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x \approx x$ 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?

## A1

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)))
```

| | |
|---|---|
| -0.39980345741334 | 5 |

`p` is evaluated 5 times.

## Q2

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?

## A2

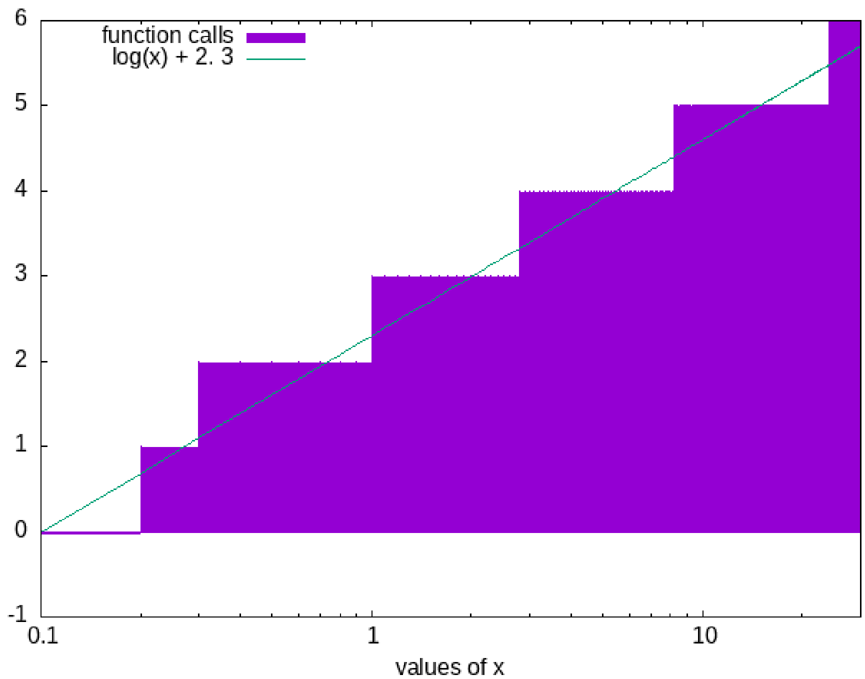
```
(use-srfis '(1))
<<1-15-p-measure>>
(let* ((vals (iota 300 0.1 0.1))
       (sines (map (λ (i)
                     (cdr (sine i)))
                   vals)))
  (zip vals sines))
```

Example output:

| | |
|---|---|
| 0.1 | 0 |
| 0.2 | 1 |
| 0.30000000000000004 | 2 |
| 0.4 | 2 |
| 0.5 | 2 |
| 0.6 | 2 |
| 0.7000000000000001 | 2 |
| 0.8 | 2 |
| 0.9 | 2 |

| | |
|---|---|
| 1.0 | 3 |

```
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

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