Hannah Zhang

1.8, 1.9, 1.10, 1.14, 1.16

```
; answers in black
; comments in yellow
; prompt in red
; question in blue
```

1.8

Implement a cube-root procedure analogous to the square-root procedure using a formula

```
; defined cube procedure for easy use later
(define (cube x)
  (* x x x))

; in sqrt, we squared the guess
; in cube root, we do guess to the power of 3
(define (good-enough? guess x)
  (< (abs (- (cube guess) x)) 0.001))

; in improve, use newton's method for cube roots
; y is an approximation and x is the value
; instead of finding the average, use the formula
(define (improve guess x)
  (/ (+ (/ x (* guess guess)) (* 2 guess)) 3))

; copied from sqrt-iter in the textbook
(define (newton guess x)
  (if (good-enough? guess x)
      guess
      (newton (improve guess x) x)))

; change newton into an easily used function
(define (cube-root x)
  (newton 1.0 x))
```

1.9

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

```
; recursive
; always has another operation after the recursive call
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))
; substitution model
; each arrow represents another call of the function “inc”
(+ 4 5)
> (+ 3 5)
> > (+ 2 5)
> > > (+ 1 5)
> > > > (+ 0 5)
< < < < 5
< < < 6
< < 7
< 8
9
```

```
; iterative
; does not have another operation after the call to itself
(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
; substitution model
> (+ 4 5)
> (+ 3 6)
> (+ 2 7)
> (+ 1 8)
> (+ 0 9)
9
```

1.10

Find the values of the following expressions and evaluate the others

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
; 2 ^ 10
(A 1 10)
→ 1024

; 2 ^ 16
(A 2 4)
→ 65536

; 2 ^ 16
(A 3 3)
→ 65536

; when x is zero, there is no recursion, it outputs 2y
(define (f n) (A 0 n))
→ 2n

; use a table, the common pattern is the exponent
```

x	n	y
1	1	2
1	2	4
1	3	8

```
(define (g n) (A 1 n))
→ 2^n

; use a table, use the computer, find a common pattern
```

x	n	y

2	1	2
2	2	4
2	3	16
2	4	65536

```
(define (h n) (A 2 n))
```

→ $2^{(h(n-1))}$

```
; work on finding the common pattern
(h 1) = (A 2 1) = 2
(h n) = (A 2 n) → (A 1 (A 2 (n - 1))
     ; from (g n) we know (A 1 n) is equal to 2^n
(h n) = 2 ^ (A 2 (n - 1))
     ; we know that (h (n-1)) = (A 2 (n - 1)), therefore
(h n) = 2 ^ (h (n - 1))

; not a recursive procedure
(define (k n) (* 5 n n))
→ 5(n^2)
```

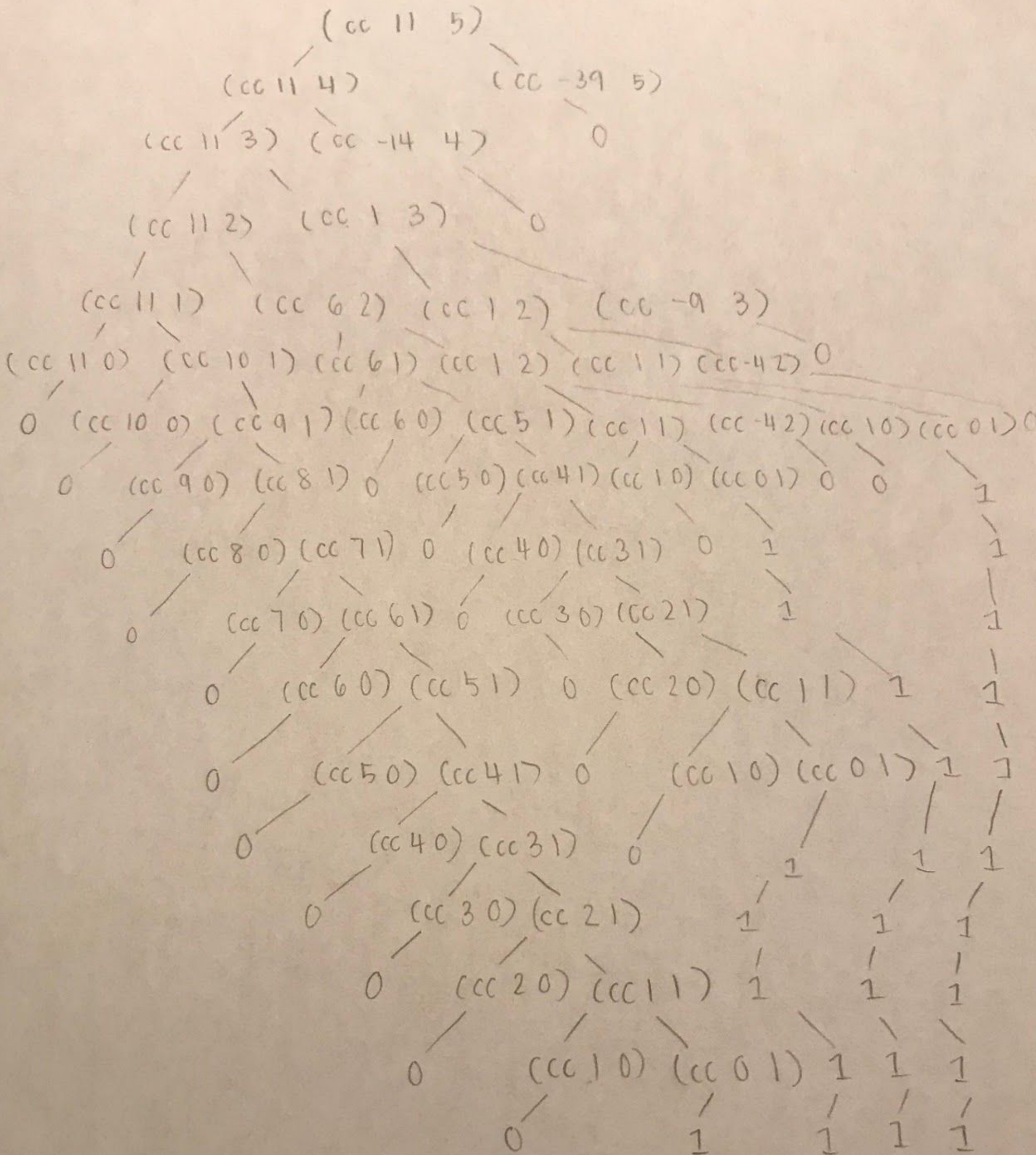
1.14

Draw the tree illustrating the process generated by the count-change procedure

```
; this function sets the amount of different coins as 5
(define (count-change amount)
  (cc amount 5))
; this function recursively calls itself to determine how many
different ways "amount" can be given back with the amount of
different coins given
(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)))))
; helper function to the previous one. determines if a certain
coin can be used or not
(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)))

; in this example we are finding the different ways to give back
11 cents with 5 possible types of coins
(cc 11 5)
```

- Order of growth: exponential
- Θ(n^2)
- Number of steps: 15 (cc is called in 15 rows)


(cc 11 5)
(cc 11 4) (cc -39 5)
(cc 11 3) (cc -14 4) 0
(cc 11 2) (cc 1 3) 0
(cc 11 1) (cc 6 2) (cc 1 2) (cc -9 3)
(cc 11 0) (cc 10 1) (cc 6 1) (cc 1 2) (cc 1 1) (cc -4 2) 0
0 (cc 10 0) (cc 9 1) (cc 6 0) (cc 5 1) (cc 1 1) (cc -4 2) (cc 1 0) (cc 0 1) 0
0 (cc 9 0) (cc 8 1) 0 (cc 5 0) (cc 4 1) (cc 1 0) (cc 0 1) 0 0 1
0 (cc 8 0) (cc 7 1) 0 (cc 4 0) (cc 3 1) 0 1 1
0 (cc 7 0) (cc 6 1) 0 (cc 3 0) (cc 2 1) 1 1
0 (cc 6 0) (cc 5 1) 0 (cc 2 0) (cc 1 1) 1 1
0 (cc 5 0) (cc 4 1) 0 (cc 1 0) (cc 0 1) 1 1
0 (cc 4 0) (cc 3 1) 0 1 1 1
0 (cc 3 0) (cc 2 1) 1 1 1
0 (cc 2 0) (cc 1 1) 1 1 1
0 (cc 1 0) (cc 0 1) 1 1 1
0 1 1 1 1

1.15

```
(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)))))
```

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

```
; when traced, this is how the recursive procedure is called
> p(sin(4.05))
> > p(sin(1.35))
> > > p(sin(0.45))
> > > > p(sin(0.15))
> > > > > p(sin(0.05))
> > > > > p(0.05)
> > > > p(0.1495)
> > > p(0.4351)
> > p(0.9758)
> p(-0.7892)
-0.40134
```

The procedure p is applied 5 times

b) 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?

- We need 5 steps to solve the problem when angle is 12.5
  - In this case, the same number of times the recursive procedure is called
- The order of growth is linear or Θ(n)
  - We know it is linear because each time, angle is divided by 3
  - Doubling the size will double the amount of times angle is divided by 3
  - This is a linear relationship

1.16

Design a procedure that turns the recursive procedure fast-expt into an iterative procedure. Use an invariant quantity and $(b^{n/2})^2 = (b^2)^{n/2}$.

```
; recursive version of fast-expt
; b^n = (b^(n/2))^2 if n is even
; b^n = b * b^(n-1) if n is odd
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
(define (even? n)
  (= (remainder n 2) 0))

; trace recursive version
(fast-expt 3 5)
> (* 3 (fast-expt 3 4))
> > (square (fast-expt 3 2))
> > > (square (fast-expt 3 1))
> > > > (* 3 (fast-expt 3 0))
; (* 3 (fast-expt 3 0) returns 1 * 3 which equals 3
< < < < 3
< < < 9
< < 81
< 243
243
```

- This version is recursive because square and b are called after fast-expt
  - n/2 is called before square
- To make it iterative, fast-expt must be called last
  - Square is called before n/2

```
; the iterative version uses an invariant quantity (a)
     ; remains unchanged from state to state
```

```
; this is how the iterative version is traced (table)
```

a	b	n
1	2	10
1	4	5
4	4	4
4	16	2
4	256	1
1024	256	0

```
; using information for how it is traced, write the function
     ; the new formula for even is given
     ; odd stays the same except the *b is changed to the
invariant quantity

; iterative version of fast-expt
(define (fast-expt2 a b n)
  (cond ((= n 0) a)
        ((even? n) (fast-expt2 a (square b) (/ n 2)))
        (else (fast-expt2 (* a b) b (- n 1)))))
```