CS61A Notes – Week 3: Applicative vs. normal order, recursive vs. iterative processes, orders of growth

Recursive vs. Iterative Processes

QUESTIONS: Will the following generate a recursive process, an iterative process, or neither?

IN CLASS:

1.
```
(define (foo x)
     (* (- (+ (/ x 3) 4) 6) 2))
```
It's not recursive, so it's pretty pointless to ask what kind of process it generates.

2.
```
(define (foo x)
    (if (= x 0) 0 (+ x (foo (- x 1)))))
```
Recursive

3.
```
(define (helper1 x)
    (if (= x 0) 1 (helper1 (- x 1))))
```
⇨ Iterative

```
(define (helper2 x)
    (if (= x 0) 1 (+ 1 (helper2 (- x 1)))))
```
=> Recursive

a.
```
(define (bar x)
     (if (even? x) (helper1 (- x 1)) (helper1 (- x 2))))
```
Iterative

b.
```
(define (bar x)
     (if (even? x) (helper2 (- x 1)) (helper2 (- x 2))))
```
Recursive

c.
```
(define (bar x)
     (cond ((= x 0) 1)
           ((= (helper2 x) 3) 5)
           (else (helper1 x))))
```
Recursive

Yoshimi Battles the Recursive Robots, Pt. 2

QUESTIONS

IN CLASS:

1. **Implement `(ab+c a b c)` that takes in values a, b, c and returns `(+ (* a b) c)`. However, you cannot use `*`. Make it a recursive process.**

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

Yes, this assumes b is positive. So sue me. What should you do if b is negative?

2. **Implement `(ab+c a b c)` as an iterative process. (If you can, do it without defining helper procedures!)**

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

3. **I want to go up a flight of stairs that has `n` steps. I can either take 1 or 2 steps each time. How many ways can I go up this flight of stairs? Write a procedure `count-stair-ways` that solves this for me.**

```
(define (count-ways n)
    (cond ((= n 0) 1)
          ((< n 0) 0)
          (else (+ (count-ways (- n 1))
                   (count-ways (- n 2))))))
```

EXTRA PRACTICE:

4. **Last week we handled something called a "falling factorial". `(falling n k)` means that `k` consecutive numbers should be multiplied together, starting from `n` and working downward. For example, `(falling 7 3)` means `7 * 6 * 5`. <u>This time, write falling-factorial *iteratively*.</u>**

```
(define (falling b n)
    (define (helper b n ans)
        (if (= n 1)
            (* b ans)
            (helper (- b 1) (- n 1) (* b ans))))
    (helper b n 1))
```

5. **Consider the `subset-sum` problem: you are given a sentence of integers and a number `k`. Is there a subset of the sentence that add up to `k`? For example,**
`(subset-sum '(2 4 7 3) 5) => #t, since 2+3=5`
`(subset-sum '(1 9 5 7 3) 2) => #f, since no combination of numbers will add to 2`

```
(define (subset-sum? nums k)
    (cond ((= k 0) #t)
          ((empty? nums) #f)
          ((< k 0) #f)
          (else (or (subset-sum? (bf nums) k)
                (subset-sum? (bf nums) (- k (first nums)))))))
```

6. **I'm standing at the origin of some x-y coordinate system for no reason when a pot of gold dropped onto the point (x, y). I would love to go get that gold, but because of some arbitrary constraints or (in my case) mental derangement, I could only move right or up one unit at a time on this coordinate system. I'd like to find out how many ways I can reach (x, y) from the origin in this fashion (because, umm, my mother asked). Write `count-ways` that solves this for me.**

```
(define (count-ways x y)
    (cond ((or (= x 0) (= y 0)) 1)
          (else (+ (count-ways x (- y 1))
                   (count-ways (- x 1) y)))))
```

Orders of Growth

QUESTIONS: What is the order of growth in time for:

IN CLASS:

1.
```
(define (fact x)
    (if (= x 0)
        1
        (* x (fact (- x 1)))))
```
Θ(n), since we subtract 1 from x each time

2.
```
(define (fact-iter x answer)
    (if (= x 0)
        answer
        (fact-iter (- x 1) (* answer x))))
```
Θ(n)

3.
```
(define (sum-of-facts n)
    (if (= n 0)
        0
        (+ (fact n) (sum-of-facts (- n 1)))))
```
Θ(n^2), since we call sum-of-facts n times, and each time we have to calculate (fact n)

4.
```
(define (fib n)
    (if (<= n 1)
        1
        (+ (fib (- n 1)) (fib (- n 2)))))
```
Θ(2^n), since we make two recursive calls each time (draw out the recursion tree and convince yourself)

5.
```
(define (foo n)
    (if (< n 1)
        0
        (+ 1 (foo (/ n 2)))))
```
Θ(log n); we cut down the input size by half each time

6.
```
(define (mod-7 n)
    (if (= (remainder n 7) 0)
        0
        (+ 1 (mod-7 (- n 1)))))
```
Θ(1); constant time! Even though we are calling it recursively, the greatest number of recursive calls is bounded by seven, and thus no matter how large your input is, you will have between 0-7 recursive calls.

EXTRA PRACTICE:

7.
```
(define (square n)
    (cond ((= n 0) 0)
          ((even? n) (* (square (quotient n 2)) 4))
          (else (+ (square (- n 1)) (- (+ n n) 1)))))
```
Θ(log n); we cut down the input size by half each time it's even. When it's odd, we make one extra recursive call, but then, once we do (- n 1), it's even again, and we get to cut it in half.