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

Recursive vs. Iterative Processes

General notes to keep in mind:

- Don't confuse "recursive procedures" and "recursive processes". A "recursive procedure" refers to the simple fact that a procedure calls itself somewhere within its body. A "recursive process" refers to the fact that the space a procedure occupies *as it runs* grows – it needs to remember additional state as it recurses. The former certainly does not imply the latter; therefore, a recursive procedure can generate an iterative process.
- The basic difference between recursive and iterative processes is that, for an iterative process, after some recursive call reaches the base case and returns, **there is nothing left to be done**. For a recursive process, after some recursive call reaches the base case and returns, **there is still work to do**. For example, for `(factorial 4)`, after reaching the base case, you still need to apply the chain of multiplications `(* 4 (* 3 (* 2 1)))`.
- Here's how you can tell: if the **last thing** a function does is make the recursive call, then the function should be an **iterative process**. If the function still has things to do after the recursive call, then it is a **recursive process**.
- Understand the `fib-iter` procedure provided in *SICP* on page 39, and be able to tell why it is more efficient than the way we used to do `fib`.

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

IN CLASS:

1.
```
(define (foo x)
     (* (- (+ (/ x 3) 4) 6) 2))
```

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

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

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

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

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

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

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.**

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

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.**

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 *iterative*ly.</u>**

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
```

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.**

Orders of Growth

When we talk about the efficiency of a procedure (at least for now), we're often interested in how much more expensive it is to run the procedure with a larger input. That is, as the size of the input grows, how do the speed of the procedure and the space its process occupies grow?

For expressing all of these, we use what is called the Big-Theta notation. For example, if we say the running time of a procedure `foo` is in $\Theta(n^2)$, we mean that the time it takes to process the input grows as the square of the size of the input. Here are some common orders of growth:

- $\Theta(1)$: constant time (takes the same amount of time regardless of input size)
- $\Theta(\log n)$: logarithmic time
- $\Theta(n)$: linear time
- $\Theta(n^2)$, $\Theta(n^3)$, etc. : polynomial time
- $\Theta(2^n)$: exponential time ("intractable"; very, very bad)

The idea of orders of growth also extends to *space*. Space refers to how much information a process must remember before it can complete. If you'll recall, the advantage of an iterative process is that it does not have to keep any information on the stack as it executes. So an iterative process always occupies a constant amount of space: $\Theta(1)$.

For a recursive process, the space it occupies tends to grow with the number of recursive calls. For example, for the factorial procedure, every time before we make a recursive call, we need to "remember" to multiply `(fact (- n 1))` by `n`. Since we'll make `n` recursive calls, we'll need to remember `n` such facts. So the orders of growth in space for the factorial function is $\Theta(n)$.

In CS61A, we're going to be mostly concerned with *time*. Time basically refers to the number of recursive calls. Intuitively, the more recursive calls we make, the more time it takes to execute the function.

A few things to note:

- If the function contains only primitive procedures like `+` or `*`, then it is constant time – $\Theta(1)$. An example would be `(define (plusone x) (+ x 1))`
- If the function is recursive, you need to:
  1. Count the number of recursive calls there will be given input `n`
  2. Count how much time it takes to process the input *per recursive call*

  The answer is usually the product of the above two. For example, given a fruit basket with `10` apples, how long does it take for me to process the whole basket? Well, I'll recursively call my `eat` procedure which eats one apple at a time (so I'll call the procedure 10 times). Each time I `eat` an apple, it takes me `30` minutes. So the total amount of time is just `30*10 = 300` minutes!
- If the function contains calls of helper functions that are not constant-time, then you need to take the orders of growth of the helper functions into consideration as well. In general, how much time the helper function takes would be factored into #2 above. (If this is confusing – and it is – try the examples below.)
- When we talk about orders of growth, we don't really care about constant factors. So if you get something like $\Theta(1000000n)$, this is really $\Theta(n)$. We can also usually ignore lower-order terms. For example, if we get something like $\Theta(n^3 + n^2 + 4n + 399)$, we take it to be $\Theta(n^3)$.

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

IN CLASS:

1.
```
(define (fact x)
    (if (= x 0)
        1
        (* x (fact (- x 1)))))
```

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

3.
```
(define (sum-of-facts n)
    (if (= n 0)
        0
        (+ (fact n) (sum-of-facts (- n 1)))))
```

4.
```
(define (fib n)
    (if (<= n 1)
        1
        (+ (fib (- n 1)) (fib (- n 2)))))
```

5.
```
(define (foo n)
    (if (< n 1)
        0
        (+ 1 (foo (/ n 2)))))
```

6.
```
 (define (mod-7 n)
    (if (= (remainder n 7) 0)
        0
        (+ 1 (mod-7 (- n 1)))))
```

EXTRA PRACTICE:

7.
```
(define (square n)
    (cond ((= n 0) 0)
          ((even? n) (* (square (quotient n 2)) 4))
          (else (+ (square (- n 1)) (- (+ n n) 1)))))
```