CS61A Notes 03 – Efficiency (and Applicative vs. Normal) [Solutions v1.0]

Applicative vs. Normal Order

QUESTIONS

1. **Above, applicative order was more efficient. Define a procedure where normal order is more efficient.**

   Anything where not evaluating the arguments will save time works. Most trivially,

   ```
   (define (f x) 3) ;; a function that always returns 3
   ```

   When you call (f (fib 10000)), applicative order would choke, but normal order would just happily drop (fib 10000) and just return 3.

2. **Evaluate this expression using both applicative and normal order: `(square (random x))`. Will you get the same result? Why or why not?**

   Unless you're lucky, the result will be quite different. Expanding to normal order, you have (* (random x) (random x)), and the two separate calls to random will probably return different values.

3. **Consider a magical function `count` that takes in no arguments, and each time it is invoked, it returns 1 more than it did before, starting with 1. Therefore, `(+ (count) (count))` will return 3. Evaluate `(square (square (count)))` with both applicative and normal order; explain your result.**

   For applicative order, (count) is only called once – returns 1 – and is squared twice. So you have (square (square 1)), which evaluates to 1.

   For normal order, (count) is called FOUR times:

   ```
   (* (square (count)) (square (count))) ==>
   (* (* (count) (count)) (* (count) (count))) ==>
   (* (* 1 2) (* 3 4)) ==>
   24
   ```

Recursive vs. Iterative Processes

QUESTIONS: Will the following generate a recursive or iterative process?

1. ```
   (define (foo x)
               (* (- (+ (/ x 3) 4) 6) 2))
   ```
   It's not a recursive procedure, 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)
                          (if (= x 0) (helper2 x) (helper1 x)))
      Iterative (when x is 0, (helper2 0) returns immediately!)

   d. (define (bar x)
                          (if (= x 0) (helper1 x) (helper2 x)))
      Recursive

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

   f. (define (bar x)
         (helper2 (helper1 x)))

      Recursive
```

Yoshimi Battles The Recursive Robots, Pt. 2

1. **There is 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`. Write the procedure falling that generates an iterative process.**

```
(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))
```

2. **Write a version of `(expt base power)` that works with negative powers as well.**

```
(define (expt base power)
                 (cond ((= power 0) 1)
                       ((> power 0) (* base (expt base (- power
1))))
                       (else (/ (expt base (+ power 1)) base))))
```

3. **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. (The problem ripped from Greg's notes)**

```
(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?
```

4. **Implement `(ab+c a b c)` as an iterative process. Don't define helper procedures.**

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

Orders of Growth

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

1. **`(define (fact x)`**

```
(define (fact x)
      (if (= x 0)
      1
      (* x (fact (- x 1)))))
Time: O(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))))
Time: O(n)
```

3.

```
(define (sum-of-facts x n)
      (if (= n 0)
            0
            (+ (fact x) (sum-of-facts x (- n 1)))))
Time: O(xn), since we have to call sum-of-facts n times, and each time
we have to calculate (fact x), which takes O(x)
```

4.

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

5.

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

Time: O(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.

6.
```
(define (gcd x y)        <======= This is hard!
                (if (= y 0)
                      x
                      (gcd y (remainder x y))))
```

Time: O(log n); we cut down the size of "x" by half in at most two recursive calls. First, note that every time we make a recursive call, we put y as the "new" x. There are two cases:

1. y < x/2; then, obviously, in the next recursive call, the "new" x (which will be y) will be less than x/2.
2. x/2 < y < x; then, in two recursive calls, the "new" x (which will be (remainder x y)) will be less than x/2. Think about that carefully: if x/2 < y, then (remainder x y) < x/2.

Don't worry if you don't get the above; it's kind of out of this course's scope. You'll learn all about it in CS70.