CS61A Week 3 | Efficiency, Higher Order Functions (v1.1)

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 $\Theta\left(n^2\right)$, we mean that the time it takes to process the input grows as the square of the size of the input. More generally, we can say that `foo` is in some $\Theta(f(n))$ if there exists some constants k_1 and k_2 and some constants c_1 and c_2 such that,

$k_1 \times f(n) <$ running time of `foo` for all $n > c_1$, and
$k_2 \times f(n) >$ running time of `foo` for all $n > c_2$

To prove, then, that foo is in $\Theta(f(n))$, we only need to find constants k_1 and k_2 where the above holds. Fortunately for you, in 61A, we're not that concerned with rigor, and you probably won't need to know exactly how to do this (you will get the painful details in 61B!) What we want you to have in 61A, then, is the intuition of guessing the orders of growth for certain procedures.

Kinds of Growth

Here are some common ones: $\Theta(1)$ - constant time (takes the same amount of time irregardless of input size); $\Theta(log(n))$ - logarithmic time; $\Theta(n)$ - linear time; $\Theta\left(n^2\right)$, $\Theta\left(n^3\right)$, etc - polynomial time; $\Theta(2^n)$ - exponential time ("intractable"; these are really, really horrible).

Orders of Growth in 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 $\Theta(n)$.

Orders of Growth in Time

"Time", for us, basically refers to the number of function calls. Intuitively, the more function 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 $30 \times 10 = 300$ 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 $\Theta(n)$. (Why? Can you use the definition of the Big-Theta notation given above to prove this?)

- We can also usually ignore lower-order terms. For example, if we get something like $\Theta\left(n^3 + n^2 + 4n + 399\right)$, we take it to be $\left(n^3\right)$. (Again, why?)

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

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 x n)
        (if (= n 0)
            0
            (+ (fact x) (sum-of-facts x (- n 1)))))
```

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

5.
```
(define (square n)
    (cond ((= n 0) 0)
        ((even? n) (* (square (quotient n 2)) 4))
        (else (+ (square (- n 1)) (- (+ n n) 1)))))
```

Procedures As Arguments

A procedure which takes other procedures as arguments is called a **higher-order procedure**. You've already seen a few examples in lecture, so I won't point them out again; let's work on something else instead. Suppose we'd like to square or double every word in the sentence:

```
(define (square-every-word sent)
        (if (empty? sent)
            '()
            (se (* (first sent) (first sent)) (square-every-word (bf sent)))))

(define (double-every-word sent)
        (if (empty? sent)
            '()
            (se (* 2 (first sent)) (double-every-word (bf sent)))))
```

Note that the only thing different about `square-every-word` and `double-every-word` is just what we do to `(first sent)`! Wouldn't it be nice to generalize procedures of this form into something more convenient? When we pass in the sentence, couldn't we specify, also, what we want to do to each word of the sentence?

To do that, we can define a higher-order procedure called `every`. `every` takes in the procedure you want to apply to each element as an argument, and applies it indiscriminately. So to write `square-every-word`, you can simply do:

```
(define (square-every-word sent) (every (lambda(x) (* x x)) sent))
```

Now isn't that nice. Nicer still: the implementation of `every` is left as a homework exercise! You should be feeling all warm and fuzzy now.

The secret to working with procedures as values is to *keep the domain and range of procedures straight!* Keep careful track of what each procedure is supposed to take in and return, and you will be fine. Let's try a few:

QUESTIONS:

1. What does this guy evaluate to: `((lambda(x) (x x)) (lambda(y) 4))`

2. What about his new best friend: `((lambda(y z) (z y)) * (lambda(a) (a 3 5)))`

3. Write a procedure, `foo`, that, given the call below, will evaluate to `10`:
   `((foo foo foo) foo 10)`

Procedures As Return Values

The problem is often: write a procedure that, given _____, *return a procedure* that _____. The keywords – conveniently emphasized – is that your procedure is supposed to return a procedure. This is often done by creating a procedure using `lambda`:

```
(define (my-wicked-procedure blah) (lambda(x y z) (...)))
```

Note that the above procedure, when called, will return a *procedure* that will take in three arguments. That is the common form for such problems (of course, never rely on these "common forms" as they'll just work against you on midterms!)

QUESTIONS:

1. In lecture, you were introduced to the procedure keep, which takes in a predicate procedure and a sentence, and throws away all words of the sentence that doesn't satisfy the predicate. The code for keep was:

```
(define (keep pred? sent)
        (cond ((empty? sent) '())
              ((pred? (first sent))
               (sentence (first sent) (keep pred? (bf sent))))
              (else (keep pred? (bf sent)))))
```

Recall that Brian said to keep numbers less than `6`, this wouldn't work: `(keep (< 6) '(4 5 6 7 8))`.

(a) Why doesn't the above work?

(b) Of course, this being Berkeley, and we being rebels, we're going to promptly prove the authority figure - the Professor himself - wrong. And just like some rebels, we'll do so by cheating. Let's do a simpler version; suppose we'd like this to do what we intended: `(keep (lessthan 6) '(4 5 6 7 8))`
Define procedure `lessthan` to make this legal.

(c) Now, how would we go about making this legal: `(keep (< 6) '(4 5 6 7 8))`

2. Taken from Midterm 1, Fall 2002

   (a) Write a procedure `scrunch` take takes in a sentence and combine the words in the sentence into one word. For example,

   ```
   >(scrunch '(here there and everywhere))
   heretheareandeverywhere
   ```

   (b) Now write `word-maker` that takes in a sentence (called template) and returns a function that takes a word as argument and returns another word based on the template, but with any * in the template replaced with the argument word. For example,

   ```
   >(define plural (word-maker '(* s)))
   >(plural 'book)
   books

   >((word-maker '(re *)) 'write)
   rewrite

   >((word-maker '(* -vs- *)) 'spy)
   spy-vs-spy
   ```

   Don't use recursion; use higher-order functions.

3. We're going to play hide-and-go-seek. Let's say, a *seeker* is a procedure that takes in a sentence, and seeks out a certain word in the sentence. It returns the word if the word is found, or `#f` otherwise. For example, if we have a `4-seeker`, a seeker that seeks out the number `4`, then

   ```
   (4-seeker '(1 2 3 4 5)) ==> 4
   (4-seeker '(1 2 3)) ==> #f
   ```

   A `seeker-producer` is a procedure that takes in an element `x` and returns a procedure (a "seeker") that takes in a sentence `sent` and returns `x` if the element `x` is in the sentence `sent`, and `#f` otherwise. For example, we can define `4-seeker` above with `(define 4-seeker (seeker-producer 4))`.

(a) Make a call to `seeker-producer` to find out if `4` is in the list `'(9 3 5 4 1 0)`. `seeker-producer` is the only procedure you can use! What does it return?

(b) Implement `seeker-producer`, using the handy-dandy procedure `member?`.

```
(define (seeker-producer x)
```

(c) Implement `seeker-producer`, using an internal define, but not using `member?`.

```
(define (seeker-producer x)
        (define (helper
```

(d) Implement `seeker-producer`, not using internal defines or `member?`.

```
(define (seeker-producer x)
```

(e) Of course, it's not much of a game if we can't hide! A *hider* of a word is a procedure that takes in a sentence and "hides" the word behind an asterisk if it exists. For example, if we have a `4-hider`, a hider that hides the number `4`, then `(4-hider '(1 2 3 4 5)) ==> (1 2 3 *4 5)`

Write a procedure `hider-producer` that takes in an element `y`, and returns a procedure (a "hider") that takes in a sentence `sent` and returns the same sentence with element `y` "hidden" behind an asterisk, if it exists.

You'll probably want to use `every` to help you.

```
(define (hider-producer x)
```

(f) Oh no! Now a hider can fool your seeker! Consider this call:
`(4-seeker (4-hider '(1 2 3 4 5))) ==> #f` (make sure you know why!)

Surely you will not be outdone by yourself. Write a procedure, `super-seeker-producer` that takes in a procedure produced by `seeker-producer` (that is, a seeker), and returns a *super-seeker* that will not be fooled by hider:
`((super-seeker-producer 4-seeker) (4-hider '(1 2 3 4 5))) ==> 4`

You can use `every` if you want. You might also find these procedures useful:

```
(define (hidden? w) (equal? (first w) '*))
(define (unhide w) (if (hidden? w) (bf w) w))
```

```
(define (super-seeker-producer seeker)
```

Recursive vs. Iterative Processes

The Professor did a good job explaining this, so just a few brief points:

- 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 or iterative process?

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

2. (Taken from Spring 2004 MT1) Consider the following procedure:

```
(define (seq n)
  (define (help n k sent)
    (cond ((= n 0) sent)
          ((> k n) (help (- n 1) 1 sent))
          (else (help n (+ k 1) (se sent k)))))
  (help n 1 '()))
```

   (a) Is it an iterative or recursive process?

   (b) What value is returned by `(seq 4)`?

(c) If your answer to part (a) is iterative, rewrite the procedure to generate a recursive process and vice versa.

More Recursions

QUESTIONS:

1. 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`

2. Implement `(ab+c a b c)` that takes in values `a`, `b`, `c` and returns $(a \times b) + c$. However, you cannot use `*`. Make it a recursive process. (The problem ripped from Greg's notes)

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