Fundamentals of Computer Science I: Media Computing (CS151.02 2007F)

Analyzing Procedures

Summary: Once you develop procedures, it becomes useful to have some sense as to how efficient the procedure is. For example, when working a list of values, some procedures take a constant number of steps (e.g., `car`), some take a number of steps proportional to the length of the list (e.g., `spots.leftmost`), and some take a number of steps proportional to the square of the length of the list. In this reading, we consider how you might analyze how slow or fast a procedure is.

Contents:

Introduction

At this point in your career, you know the basic tools to build algorithms, including conditionals, recursive loops, variables, and subroutines. You've also started to write documentation to explain what your procedures do and to write test suites that help ensure that you write correct procedures.

You'll soon find that you can often write a variety of procedures that solve the same problem, all of which pass your test suite. How do you then decide which one to use? There are many criteria we use. One important one is *readability* - can we easily understand the way the algorithm works? A more readable algorithm is also easier to correct if we ever notice an error or to modify if we want to expand its capabilities.

However, most programmers care as much about *efficiency* - how many computing resources does the algorithm use? (Pointy-haired bosses care even more about such things.) Resources include memory and processing time. Most analyses of efficiency focus on running time, the amount of time the program takes to run, which, in turn depends on both how many steps the procedure executes and how long each step takes.. Since almost every step in Scheme involves a procedure call, to get a sense of the approximate running time of Scheme algorithms, we can usually count procedure calls.

In this reading and the corresponding lab, we will consider some techniques for figuring out how many procedure calls are done.

Some Examples to Explore

As we explore analysis, we'll start with a few basic examples. First, consider two versions of the `rgb-list.brightest` procedure, which you should recall from a recent lab.

```
;;; Procedure:
;;;   rgb.brightest-1
;;;   rgb.brightest-2
;;; Parameters:
;;;   colors, a list of RGB colors
;;; Purpose:
;;;   Find the brightest color in colors.
;;; Produces:
;;;   brightest, an RGB color.
;;; Preconditions:
;;;   colors is nonempty. [Unverified]
;;;   Each element of colors is an RGB color.  [Unverified]
;;; Postconditions:
;;;   brightest is equal to at least one element of colors.
;;;   brightest is at least as bright as each element of colors.
(define rgb.brightest-1
  (lambda (colors)
    (cond
      ((null? (cdr colors))
       (car colors))
      ((rgb.brighter? (car colors) (rgb.brightest-1 (cdr colors)))
       (car colors))
      (else (rgb.brightest-1 (cdr colors))))))
(define rgb.brightest-2
  (lambda (colors)
    (if (null? (cdr colors))
        (car colors)
        (rgb.brighter (car colors) (rgb.brightest-2 (cdr colors))))))
```

You can find the helpers for this procedure in an appendix.

The two versions are fairly similar. Does it matter which we use? We'll see in a bit.

As a second example, let's consider how we might write the famous `reverse` procedure ourselves, rather than using the built-in version. In this example, we'll use two very different versions.

```
;;; Procedures:
;;;   reverse-1
;;;   reverse-2
;;; Parameters:
;;;   lst, a list of size n
;;; Purpose:
;;;   Reverse lst.
;;; Produces:
;;;   reversed, a list
;;; Preconditions:
;;;   lst is a list. [Unverified]
;;; Postconditions:
;;;   For all indices i,
;;;     (list-ref lst i) equals (list-ref reversed (- n i 1))
```

```
(define reverse-1
  (lambda  (lst)
    (if (null? lst)
        null
        (my-append (reverse-1 (cdr lst)) (list (car lst))))))
(define reverse-2
  (lambda (lst)
    (reverse-2-kernel null lst)))
(define reverse-2-kernel
  (lambda (reversed remaining)
    (if (null? remaining)
        reversed
       (reverse-2-kernel (cons (car remaining) reversed)
                         (cdr remaining) ))))
```

You'll note that we've used `my-append` rather than `append`. Why? Because we know that the `append` procedure is recursive, so we want to make sure that we can count the calls that happen there, too. We've used the standard implementation of `append` in defining `my-append`.

```
;;; Procedure:
;;;   my-append
;;; Parameters:
;;;   front, a list of size n
;;;   back, a list of size m
;;; Purpose:
;;;   Put front and back together into a single list.
;;; Produces:
;;;   appended, a list of size n+m.
;;; Preconditions:
;;;   front is a list [Unverified]
;;;   back is a list [Unverified]
;;; Postconditions:
;;;   For all i, 0 <= i < n,
;;;    (list-ref appended i) is (list-ref front i)
;;;   For all i, n <= i < n+m
;;;;   (list-ref appended i) is (list-ref back (- i n))
(define my-append
  (lambda (front back)
    (if (null? front)
        back
        (cons (car front) (my-append (cdr front) back)))))
```

Strategy One: Counting Steps Through Output

One obvious way to figure out how many steps a procedure takes is to add a bit of code to output something for each procedure call. While you'll learn the details of output later, all you need to know right now is that `display` prints its parameter, and `newline` prints a carriage return. To annotate the procedures for output, we simply add appropriate calls. For example, here are the first few lines of the annotated versions of `rgb.brightest-1` and `rgb.brightest-2`.

```
(define rgb.brightest-1
  (lambda (colors)
    (display (list 'rgb-brightest-1 (map rgb->string colors))) (newline)
    (cond
```

```
      ((null? (cdr colors))
       (car colors))
      ...)))

(define rgb.brightest-2
  (lambda (colors)
    (display (list 'rgb-brightest-2 (map rgb->string colors))) (newline)
    (if (null? (cdr colors))
        (car colors)
        (rgb.brighter (car colors) (rgb.brightest-2 (cdr colors))))))
```

So, what happens when we call the two procedures on a simple list of colors?

```
> (define grey (lambda (n) (rgb.new n n n)))
> (define light-to-dark (map grey (list 255 192 128 64 0)))
> (rgb.brightest-1 light-to-dark)
-1
(rgb.brightest-1 (255/255/255 192/192/192 128/128/128 64/64/64 0/0/0))
(rgb.brightest-1 (192/192/192 128/128/128 64/64/64 0/0/0))
(rgb.brightest-1 (128/128/128 64/64/64 0/0/0))
(rgb.brightest-1 (64/64/64 0/0/0))
(rgb.brightest-1 (0/0/0))
> (rgb.brightest-2 light-to-dark)
-1
(rgb.brightest-2 (255/255/255 192/192/192 128/128/128 64/64/64 0/0/0))
(rgb.brightest-2 (192/192/192 128/128/128 64/64/64 0/0/0))
(rgb.brightest-2 (128/128/128 64/64/64 0/0/0))
(rgb.brightest-2 (64/64/64 0/0/0))
(rgb.brightest-2 (0/0/0))
```

So far, so good. Each has about five calls for a list of length five. Let's try a slightly different list.

```
> (define dark-to-light (reverse light-to-dark))
> (rgb.brightest-1 dark-to-light)
-1
(rgb.brightest-1 (0/0/0 64/64/64 128/128/128 192/192/192 255/255/255))
(rgb.brightest-1 (64/64/64 128/128/128 192/192/192 255/255/255))
(rgb.brightest-1 (128/128/128 192/192/192 255/255/255))
(rgb.brightest-1 (192/192/192 255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (192/192/192 255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (128/128/128 192/192/192 255/255/255))
(rgb.brightest-1 (192/192/192 255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (192/192/192 255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (64/64/64 128/128/128 192/192/192 255/255/255))
(rgb.brightest-1 (128/128/128 192/192/192 255/255/255))
(rgb.brightest-1 (192/192/192 255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (255/255/255))
```

```
(rgb.brightest-1 (192/192/192 255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (128/128/128 192/192/192 255/255/255))
(rgb.brightest-1 (192/192/192 255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (192/192/192 255/255/255))
(rgb.brightest-1 (255/255/255))
(rgb.brightest-1 (255/255/255))
```

Wow! That's a lot of lines to count, 31 if I've counted correctly. That seems like a few more than we'd expect. That may be a problem. So, how many does the other version take?

```
> (rgb.brightest-2 dark-to-light)
-1
(rgb.brightest-2 (0/0/0 64/64/64 128/128/128 192/192/192 255/255/255))
(rgb.brightest-2 (64/64/64 128/128/128 192/192/192 255/255/255))
(rgb.brightest-2 (128/128/128 192/192/192 255/255/255))
(rgb.brightest-2 (192/192/192 255/255/255))
(rgb.brightest-2 (255/255/255))
```

Still five calls for a list of length five. Clearly, the second one is better on this input. Why and how much? That's an issue for a bit later.

In case you haven't noticed, we've now tried two special cases, one in which the list is organized brightest to darkest and one in which the list is organized darkest to brightest. In the first case, the two versions are equivalent. In the second, the second implementation is significantly faster. We should certainly try some others. As in the case of unit testing, the cases you test for efficiency matter a lot.

Now, let's try the other example. Say let's reverse a list of length five.

```
> (reverse-1 (list 1 2 3 4 5))
(5 4 3 2 1)
(reverse-1 (1 2 3 4 5))
(reverse-1 (2 3 4 5))
(reverse-1 (3 4 5))
(reverse-1 (4 5))
(reverse-1 (5))
(reverse-1 ())
> (reverse-2 (list 1 2 3 4 5))
(5 4 3 2 1)
(reverse-2 (1 2 3 4 5))
(reverse-2-kernel (1 2 3 4 5) ())
(reverse-2-kernel (2 3 4 5) (1))
(reverse-2-kernel (3 4 5) (2 1))
(reverse-2-kernel (4 5) (3 2 1))
(reverse-2-kernel (5) (4 3 2 1))
(reverse-2-kernel () (5 4 3 2 1))
```

So far, so good. The two seem about equivalent. But what about the other procedures that each calls? The `kernel` of `reverse-2` calls `cdr`, `cons`, and `car` once for each recursive call. Hence, there are probably five times as many procedure calls as we just counted. On the other hand, `reverse-1` calls `my-append` and `list`. The `list` procedure is not recursive, so we don't need to worry about it. But

what about `my-append`? It is recursive, so let's add an output annotation to that procedure, too. Now, what happens?

```
> (reverse-1 (list 1 2 3 4 5))
(reverse-1 (1 2 3 4 5))
(reverse-1 (2 3 4 5))
(reverse-1 (3 4 5))
(reverse-1 (4 5))
(reverse-1 (5))
(reverse-1 ())
(my-append () (5))
(my-append (5) (4))
(my-append () (4))
(my-append (5 4) (3))
(my-append (4) (3))
(my-append () (3))
(my-append (5 4 3) (2))
(my-append (4 3) (2))
(my-append (3) (2))
(my-append () (2))
(my-append (5 4 3 2) (1))
(my-append (4 3 2) (1))
(my-append (3 2) (1))
(my-append (2) (1))
(my-append () (1))
(5 4 3 2 1)
```

Hmmm, that's a few calls to `append`.. Not a ton, but some. Let's see ... fifteen, if we count correctly. Now, what happens when we add one element to the list.

```
> (reverse-1 (list 1 2 3 4 5 6))
(reverse-1 (1 2 3 4 5 6))
(reverse-1 (2 3 4 5 6))
(reverse-1 (3 4 5 6))
(reverse-1 (4 5 6))
(reverse-1 (5 6))
(reverse-1 (6))
(reverse-1 ())
(my-append () (6))
(my-append (6) (5))
(my-append () (5))
(my-append (6 5) (4))
(my-append (5) (4))
(my-append () (4))
(my-append (6 5 4) (3))
(my-append (5 4) (3))
(my-append (4) (3))
(my-append () (3))
(my-append (6 5 4 3) (2))
(my-append (5 4 3) (2))
(my-append (4 3) (2))
(my-append (3) (2))
(my-append () (2))
(my-append (6 5 4 3 2) (1))
(my-append (5 4 3 2) (1))
(my-append (4 3 2) (1))
```

```
(my-append (3 2) (1))
(my-append (2) (1))
(my-append () (1))
(6 5 4 3 2 1)
> (reverse-2 (list 1 2 3 4 5 6))
(reverse-2 (1 2 3 4 5 6))(kernel (1 2 3 4 5 6) ())
(kernel (2 3 4 5 6) (1))
(kernel (3 4 5 6) (2 1))
(kernel (4 5 6) (3 2 1))
(kernel (5 6) (4 3 2 1))
(kernel (6) (5 4 3 2 1))
(kernel () (6 5 4 3 2 1))
(6 5 4 3 2 1)
```

Hmmm ... we added one element to the list, and we added six calls to `my-append` (we're now up to 21). In the case of `reverse-2`, we seem to have added only one call.

What if there are ten elements? We're not sure we want to count that high, but we're pretty sure we now have 55 total calls to `my-append`, and only ten recursive calls to the kernel.

Strategy Two: Automating the Counting of Steps

Okay, we've seen a few problems in the previous strategy. First, it's a bit of a pain to add the output annotations to our code. Second, it's even more of a pain to count the number of lines those annotations produce. Finally, there are some procedure calls that we didn't count. So, what should we do? In some sense, we want to make the same transition here that we made in doing testing - from manual testing to automatic testing.

Some of the Grinnell faculty have built a simple library that helps you make that transition. How do you use that library? It's relatively straightforward, but still requires you to modify your code a bit. In particular, for any procedure of interest, replace `define` with `define$`. In this case, we might write

```
(define$ rgb.brightest-1
  (lambda (colors)
    ...))
```

Finally, to report on all the counted procedure calls made during the evaluation of a particular expression, we write

```
(analyze expression proc)
```

For example, we can redo our first analyses as follows:

```
> (analyze (rgb.brightest-1 light-to-dark) rgb.brightest-1)
-1
rgb.brightest-1: 4
Total: 4
> (analyze (rgb.brightest-2 light-to-dark) rgb.brightest-2)
-1
rgb.brightest-2: 4
Total: 4
```

Now, we can try the second analysis of the `rgb-brightest` procedures and even do a bit less counting.

```
> (analyze (rgb.brightest-1 dark-to-light) rgb.brightest-1)
-1
rgb.brightest-1: 30
Total: 30
> (analyze (rgb.brightest-2 dark-to-light) rgb.brightest-2)
-1
rgb.brightest-2: 4
Total: 4
```

What if we try a slightly bigger list? We didn't really want to do so when we were counting by hand, but now the computer can count for us.

```
> (define lots-of-greys (map grey (list 0 16 32 48 64 96 112 128 144 160 176 192 208 224 240 255)))
> (analyze (rgb.brightest-1 lots-of-greys) rgb.brightest-1)
-1
rgb.brightest-1: 65534
Total: 65534
```

What about the other procedures involved (`null?`, `cdr`, etc)? The testing library provides a variant of the `analyze` routine in which you do not list the procedures to count. In this case, it counts as many as it can.

```
> (analyze (rgb.brightest-1 lots-of-greys))
-1
car: 65535
cdr: 131069
null?: 65535
rgb.brighter?: 32767
rgb.brightest-1: 6553
Total: 360440
> (analyze (rgb.brightest-2 lots-of-greys))
-1
car: 16
cdr: 31
null?: 16
rgb.brighter: 15
rgb.brightest-2: 15
Total: 93
```

Which one would you prefer to use?

You may be waiting to analyze the two forms of `reverse`, but we'll leave that as a task for the lab.

Interpreting Results

You now have a variety of ways to count steps. But what should you do with those results in comparing algorithms? The strategy most computer scientists use is to look for patterns that describe the relationship between the size of the input and the number of procedure calls. I find it useful to look at what happens when you double the input size (e.g., go from a list of length 4 to a list of length 8, or go from the number 8 to the number 16).

The most efficient algorithms generally use a number of procedure calls that does not depend on the size of the input. For example, `car` always takes one step, whether the list of length 1, 2, 4, or even 1024.

At times, we'll find algorithms that add a constant number of steps when you double the input size (we haven't seen any of those so far). Those are also very good.

However, the best we normally do are the algorithms that take about twice as many steps when we double the size of the input. For example, in processing a list of length n, in a situation in which we expect to look at every element of the list, we'll probably do a few steps for each element. If we double the list, we double the number of steps. Most of the list problems you face in this class should have this form.

A few times, you'll notice that when you double the input, the number of steps seems to go up by about a factor of four. Such algorithms are sometimes necessary, but get slow.

At times, you'll find that when you double the input size, the number of steps goes up much more than a factor of four. (For example, that happens in the first version of `rgb.brightest-1`.) If you find your code exhibiting that behavior, it's time to write a new algorithm.

What Went Wrong?

So, why are the slower versions of the two algorithms above so slow? In the case of `rgb.brightest-1`, there may be two identical recursive calls for each call. That doubling gets painful fairly quickly. If you notice that you are doing multiple recursive calls, see if you can apply a helper to the recursive call, as we did in `rgb-list.brightest-2`. Rewriting your procedure using the primary tail recursion strategy (that is, accumulating a result as you go) may also be helpful.

In the case of `reverse-1`, the difficulty is only obvious when we include `my-append`. And what's the problem? The problem is that `my-append` is not a constant-time algorithm, but one that needs to visit every element in the first list. Since `reverse` keeps calling it with larger and larger lists, we spend a lot of time appending.

Appendix: Utilities for Computing with Brightness

At the beginning of this reading, we considered two techniques for computing the brightest element of a list. Each of those techniques relied on some additional procedures, such as `rgb.brighter?` and `rgb.brighter`. Those procedures, and other related procedures, may be found here.

```
;;; Procedure:
;;;   rgb.brightness
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Computes the brightness of color on a 0 (dark) to 100 (bright) scale.
;;; Produces:
;;;   b, an integer
;;; Preconditions:
;;;   color is a valid RGB color.  That is, each component is between
;;;     0 and 255, inclusive.
;;; Postconditions:
```

```
;;;   If color1 is likely to be perceived as brighter than color2,
;;;     then (brightness color1) > (brightness color2).
(define rgb.brightness
  (lambda (color)
    (round (* 100 (/ (+ (* .30 (rgb.red color))
                        (* .59 (rgb.green color))
                        (* .11 (rgb.blue color)))
                     255)))))

;;; Procedure:
;;;   rgb.brighter?
;;; Parameters:
;;;   color1, an RGB color
;;;   color2, an RGB color
;;; Purpose:
;;;   Determine if color1 is brighter than color2.
;;; Produces:
;;;   brighter?, a Boolean
;;; Preconditions:
;;;   color1 and color2 are valid RGB colors. [Unverified]
;;; Postconditions:
;;;   If the brightness of color1 (as determined by rgb.brightness) is
;;;     at least as great as the brightness of color2, then brighter?
;;;     is #t.
;;;   Otherwise, brighter? is #f.
(define rgb.brighter?
  (lambda (color1 color2)
    (>= (rgb.brightness color1) (rgb.brightness color2))))

;;; Procedure:
;;;   rgb.brighter
;;; Parameters:
;;;   color1, an RGB color
;;;   color2, an RGB color
;;; Purpose:
;;;   Find the brighter of color1 and color2.
;;; Produces:
;;;   brighter, a color
;;; Preconditions:
;;;   color1 and color2 are valid RGB color. [Unverified]
;;; Postconditions:
;;;   brighter is equal to color1 or color2 (or both).
;;;   brighter is at least as bright as color1.
;;;   brighter is at least as bright as color2.
(define rgb.brighter
  (lambda (color1 color2)
    (if (rgb.brighter? color1 color2)
        color1 color2)))
```

 This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.