

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., `last-in-list`), and some take a number of steps proportional to the square of the length of the list. In this reading, we consider how you figure out how slow or fast a procedure is.

Contents:

- Introduction
- Some Examples to Explore
- Strategy One: Counting Steps Through Output
- Strategy Two: Automating the Counting of Steps
- Interpreting Results
- What Went Wrong?

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. 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, two versions of the `largest-of-list` procedure, one that does not use a local variable and one that does. (You may find this example familiar from a recent lab.)

```

;;; Procedures:
;;; largest-of-list-1
;;; largest-of-list-2
;;; Parameters:
;;; lst, a nonempty list of real numbers [unverified]
;;; Purpose:
;;; Find the largest number in lst.
;;; Produces:
;;; largest, a real number
;;; Preconditions:
;;; [No additional preconditions]
;;; Postconditions:
;;; largest is an element of lst.
;;; For all valid indices i, largest >= (list-ref lst i)
(define largest-of-list-1
  (lambda (lst)
    (if (null? (cdr lst))
        (car lst)
        (if (> (car lst) (largest-of-list-1 (cdr lst)))
            (car lst)
            (largest-of-list-1 (cdr lst))))))
(define largest-of-list-2
  (lambda (lst)
    (if (null? (cdr lst))
        (car lst)
        (let ((largest-remaining (largest-of-list-2 (cdr lst))))
          (if (> (car lst) largest-remaining)
              (car lst)
              largest-remaining))))))

```

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 [unverified]
;;; Purpose:
;;; Reverse lst.
;;; Produces:
;;; reversed, a list
;;; Preconditions:
;;; [No additional preconditions]
;;; 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
        (myappend (reverse-1 (cdr lst)) (list (car lst))))))
(define reverse-2
  (letrec ((kernel

```

```

        (lambda (remaining reversed)
          (if (null? remaining)
              reversed
              (kernel (cdr remaining) (cons (car remaining) reversed))))))
(lambda (lst)
  (kernel lst null)))

```

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

```

;;; Procedure:
;;; myappend
;;; Parameters:
;;; front, a list of size n [unverified]
;;; back, a list of size m [unverified]
;;; Purpose:
;;; Put front and back together into a single list.
;;; Produces:
;;; appended, a list of size n+m.
;;; Preconditions:
;;; [No additional preconditions]
;;; 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 myappend
  (lambda (front back)
    (if (null? front)
        back
        (cons (car front) (myappend (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. We've done that before, to keep track of what happens in some of our procedures. Here, we'll just count the output. For example, here are the first few lines of the annotated versions of `largest-in-list-1` and `largest-in-list-2`.

```

(define largest-of-list-1
  (lambda (lst)
    (display (list 'largest-of-list-1 lst))
    (newline)
    (if (null? (cdr lst))
        ...)))
(define largest-of-list-2
  (lambda (lst)
    (display (list 'largest-of-list-2 lst))
    (newline)
    (if (null? (cdr lst))
        ...)))

```

So, what happens when we call the two procedures on a simple list.

```
> (largest-of-list-1 (list 8 3 2 1 0))
(largest-of-list-1 (8 3 2 1 0))
(largest-of-list-1 (3 2 1 0))
(largest-of-list-1 (2 1 0))
(largest-of-list-1 (1 0))
(largest-of-list-1 (0))
8
> (largest-of-list-2 (list 8 3 2 1 0))
(largest-of-list-2 (8 3 2 1 0))
(largest-of-list-2 (3 2 1 0))
(largest-of-list-2 (2 1 0))
(largest-of-list-2 (1 0))
(largest-of-list-2 (0))
8
```

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

```
> (largest-of-list-1 (list 0 3 7 8 9 23))
(largest-of-list-1 (0 3 7 8 9 23))
(largest-of-list-1 (3 7 8 9 23))
(largest-of-list-1 (7 8 9 23))
(largest-of-list-1 (8 9 23))
(largest-of-list-1 (9 23))
(largest-of-list-1 (23))
(largest-of-list-1 (23))
(largest-of-list-1 (9 23))
(largest-of-list-1 (23))
fifty lines deleted
(largest-of-list-1 (23))
(largest-of-list-1 (9 23))
(largest-of-list-1 (23))
(largest-of-list-1 (23))
23
```

Wow! That's a lot of lines to count. I deleted fifty lines, and there are still 14 lines remaining. Wow! 64 procedure calls, if I counted correctly. That may be a problem. So, how many does the other version take?

```
> (largest-of-list-2 (list 0 3 7 8 9 23))
(largest-of-list-2 (0 3 7 8 9 23))
(largest-of-list-2 (3 7 8 9 23))
(largest-of-list-2 (7 8 9 23))
(largest-of-list-2 (8 9 23))
(largest-of-list-2 (9 23))
(largest-of-list-2 (23))
23
```

Only 6 calls for a list of length six. 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 largest to smallest and one in which the list is organized smallest to largest. 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))
(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 ())
(5 4 3 2 1)
> (reverse-2 (list 1 2 3 4 5))
(reverse-2 (1 2 3 4 5))
(kernel (1 2 3 4 5) ())
(kernel (2 3 4 5) (1))
(kernel (3 4 5) (2 1))
(kernel (4 5) (3 2 1))
(kernel (5) (4 3 2 1))
(kernel () (5 4 3 2 1))
(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 `myappend` and `list`. The `list` procedure is not recursive, so we don't need to worry about it. But what about `myappend`? 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 ())
(myappend () (5))
(myappend (5) (4))
(myappend () (4))
(myappend (5 4) (3))
(myappend (4) (3))
(myappend () (3))
(myappend (5 4 3) (2))
(myappend (4 3) (2))
(myappend (3) (2))
(myappend () (2))
(myappend (5 4 3 2) (1))
(myappend (4 3 2) (1))
(myappend (3 2) (1))
(myappend (2) (1))
(myappend () (1))
(5 4 3 2 1)
```

Hmmm ... that's a few calls to `append`.. Not a ton, but some. Let's see ... fifteen, if I 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 ())
(myappend () (6))
(myappend (6) (5))
(myappend () (5))
(myappend (6 5) (4))
(myappend (5) (4))
(myappend () (4))
(myappend (6 5 4) (3))
(myappend (5 4) (3))
(myappend (4) (3))
(myappend () (3))
(myappend (6 5 4 3) (2))
(myappend (5 4 3) (2))
(myappend (4 3) (2))
(myappend (3) (2))
(myappend () (2))
(myappend (6 5 4 3 2) (1))
(myappend (5 4 3 2) (1))
(myappend (4 3 2) (1))
(myappend (3 2) (1))
(myappend (2) (1))
(myappend () (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 `myappend` (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? I'm not sure I want to count that high, but I'm pretty sure we now have 55 total calls to `myappend`, 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.

First, you need to load the appropriate file, `analyst.scm`.

```
(load "/home/rebelsky/Web/Courses/CS151/2007S/Examples/analyst.scm")
```

Next, for any procedure of interest, replace `define` with `define$`. In this case, we might write

```
(define$ largest-in-list-1
  (lambda (lst)
    ...))
```

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 (largest-of-list-1 (list 8 3 2 1 0)) largest-of-list-1)
largest-of-list-1: 4
Total: 4
8
> (analyze (largest-of-list-2 (list 8 3 2 1 0)) largest-of-list-2)
largest-of-list-2: 4
Total: 4
8
```

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

```
> (analyze (largest-of-list-1 (list 0 3 7 8 9 23)) largest-of-list-1)
largest-of-list-1: 62
Total: 62
23
> (analyze (largest-of-list-2 (list 0 3 7 8 9 23)) largest-of-list-2)
largest-of-list-2: 5
Total: 5
23
```

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.

```
> (analyze (largest-of-list-1 (list 0 3 7 8 9 23 67 111)) largest-of-list-1)
largest-of-list-1: 254
Total: 254
111
> (analyze (largest-of-list-2 (list 0 3 7 8 9 23 67 111)) largest-of-list-2)
largest-of-list-2: 7
Total: 7
111
```

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 (largest-of-list-1 (list 0 3 7 8 9 23 67 111)))
>: 127
car: 255
cdr: 509
largest-of-list-1: 254
null?: 255
Total: 1400
111
> (analyze (largest-of-list-2 (list 0 3 7 8 9 23 67 111)))
>: 7
car: 8
cdr: 15
largest-of-list-2: 7
null?: 8
Total: 45
111
```

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 `greatest-of-list`.) 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 `greatest-of-list-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 use a local variable to reduce the number.

In the case of `reverse-1`, the difficulty is only obvious when we include `myappend`. And what's the problem? The problem is that `myappend` 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.

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.