exercises-1.2

Plant Kingdom

March 19, 2014

Contents

1 Section 1.2

1.1 Section 1.2.1 Linear Recursion and Iteration

1.1.1 Exercise 1.9: Thinking about procedures and processes

1. Problem Each of the following two procedures defines a method for adding two positive integers in terms of the procedures 'inc', which increments its argument by 1, and 'dec', which decrements its argument by 1.

```
(define (+ a b)
  (if (= a 0)
  b
  (inc (+ (dec a) b))))

(define (+ a b)
  (if (= a 0)
  b
  (+ (dec a) (inc b))))
```

   Using the substitution model, illustrate the process generated by each procedure in evaluating `(+ 4 5)`. Are these processes iterative or recursive?

2. Answer

   First procedure:

```
(+ 4 5)
(if (= 4 0)
    5
    (inc (+ (dec 4) 5)))
```

```
(inc (+ (dec 4) 5))
(inc (+ 3 5))
(inc (if (= 3 5)
         5
         (inc (+ (dec 3) 5))))
(inc (inc (+ (dec 3) 5)))
(inc (inc (+ 2 5) ))
(inc (inc (if (= 2 5)
              5
              (inc (+ (dec 2) 5)))))
(inc (inc (inc (+ (dec 2) 5))))
(inc (inc (inc (+ 1 5))))
(inc (inc (inc (if (= 1 0)
                   5
                   (inc (+ (dec 1) 5))))))
(inc (inc (inc (inc (+ (dec 1) 5)))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc (if (= 0 0)
                        5
                        (inc (+ (dec 0) 5)))))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
9
```

Second procedure:

```
(+ 4 5)
(if (= 4 0)
    5
    (+ (dec 4) (inc 5)))
(+ (dec 4) (inc 5))
(+ 3 6)
(if (= 3 0)
    6
    (+ (dec 3) (inc 6)))
(+ (dec 3) (inc 6))
(+ 2 7)
(if (= 2 0)
```

```
    7
    (+ (dec 2) (inc 7)))
(+ (dec 2) (inc 7))
(+ 1 8)
(if (= 1 0)
    8
    (+ (dec 1) (inc 8)))
(+ (dec 1) (inc 8))
(+ 0 9)
(if (= 0 0)
    9
    (+ (dec 0) (inc 9)))
9
```

As can be seen, the first procedure is a linear recursive process, building up deferred operations. The second is a linear iterative process.

1.1.2 Exercise 1.10: Ackermann's function

The following procedure computes a mathematical function called Ackermann's function.

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

What are the values of the following expressions?

```
(A 1 10)
```

```
(A 2 4)
```

```
(A 3 3)
```

Consider the following procedures, where `A` is the procedure defined above:

```
(define (f n) (A 0 n))

(define (g n) (A 1 n))

(define (h n) (A 2 n))

(define (k n) (* 5 n n))

(define (investigate name func num)

  (define (investigate-iter i)
    (cond ((< i num)
           (printf " ~a:~a" i (func i))
           (investigate-iter (+ i 1)))
          (else (newline))))
  (display name)
  (investigate-iter 0))


(investigate "f" f 17)
(investigate "g" g 17)
(investigate "h" h 5)
(investigate "k" k 5)
```

Give concise mathematical definitions for the functions computed by the procedures `f`, `g`, and `h` for positive integer values of n. For example, `(k n)` computes $5n^2$.

$$f(n) => 2 * n$$

$$g(n) => 2^n$$

```
(A 2 1) => 2                                          (2^(2^0))
(A 2 2) => (A 1 (A 2 1)) => (A 1 2) => 4              (2^(2^1))  (2^(2^(2^0))
(A 2 3) => (A 1 (A 2 2)) => (A 1 4) => 16             (2^(2^2))  (2^(2^(2^1))
(A 2 4) => (A 1 (A 2 3)) => (A 1 16) => 65536         (2^(2^4))  (2^(2^(2^2))
(A 2 5) => (A 1 (A 2 4)) => (A 1 65536) => 20035...56736 (2^(2^16)) (2^(2^(2^4))

2^2 ^ 2^(1)
2^3 ^ 2^2 ^ 2^1
```

```
1 => 2
2 => 2 ^ 2
3 => 2 ^ 2 ^ 2
4 => 2 ^ 2 ^ 2 ^ 2
...
```

There's no standard mathematical notation; this is a "power tower", also called *tetration*.

1.2 Section 1.2.2 Tree Recursion

1.2.1 Exercise 1.11: Converting a recursive process to an iterative process

1. Problem

   A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n >= 3$. Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

2. Answer

```
(define (f-rec n)
  (if (< n 3)
      n
      (+ (f-rec (- n 1))
         (* 2 (f-rec (- n 2)))
         (* 3 (f-rec (- n 3))))))

(define (f-iter n)

  (define (f-inner i fn1 fn2 fn3)
    (define cur (+ fn1 (* 2 fn2) (* 3 fn3)))
    (if (= i n)
        cur
        (f-inner (+ i 1) cur fn1 fn2)))

  (if (< n 3)
      n
      (f-inner 3 2 1 0)))
```

```
(define (compare f1 f2 k)

  (define (compare-iter i good?)
    (define t0 (current-milliseconds))
    (define r1 (f1 i))
    (define t1 (current-milliseconds))
    (define r2 (f2 i))
    (define t2 (current-milliseconds))

    (printf "~a: ~a (~a ms) ~a (~a ms) => ~a~n"
            i
            r1 (- t1 t0)
            r2 (- t2 t1)
            (= r1 r2))
    (if (< i k)
        (compare-iter (+ i 1) (and good? (= r1 r2)))
        (and good? (= r1 r2))))

  (compare-iter 1 true))

(compare f-rec f-iter 33)
```

1.2.2 Exercise 1.12: Computing Pascal's triangle

1. Problem The following pattern of numbers is called "Pascal's triangle".

   ```
           1
         1   1
       1   2   1
     1   3   3   1
   1   4   6   4   1
   ```

   The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

2. Answer

   ```
   ;; Find the m-th number in the n-th row of Pascal's triangle
   ```

```
(define (pascal n m)
  (cond ((or (> m n) (< m 1) (< n 1)) -1) ; Error condition, need an exception her
        ((or (= m 1) (= m n)) 1)  ; Outer numbers
        (else (+ (pascal (- n 1) (- m 1))
                 (pascal (- n 1) m)))))

;; Display first k rows of Pascal's triangle
(define (display-pascal k)
  (define (display-pascal-inner i j)
    (display (pascal i j))
    (cond ((< j i) (display " ") (display-pascal-inner i (+ j 1)))
          ((= i k) (newline))
          ((= i j) (newline) (display-pascal-inner (+ i 1) 1))))

  (display-pascal-inner 1 1))

(time (display-pascal 18))
```

1.2.3 WRITEUP Exercise 1.13: A Fibonacci proof

Prove that Fib(n) is the closest integer to $\phi^n/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2$. Hint: Let $\psi = (1-\sqrt{5})/2$. Use induction and the definition of the Fibonacci numbers (see section 1.2.2) to prove that Fib(n) $= (\phi^n - \psi^n)/\sqrt{5}$.

1.3 Section 1.2.3 Orders of Growth

1.3.1 TODO Exercise 1.14: Counting change

1. Problem

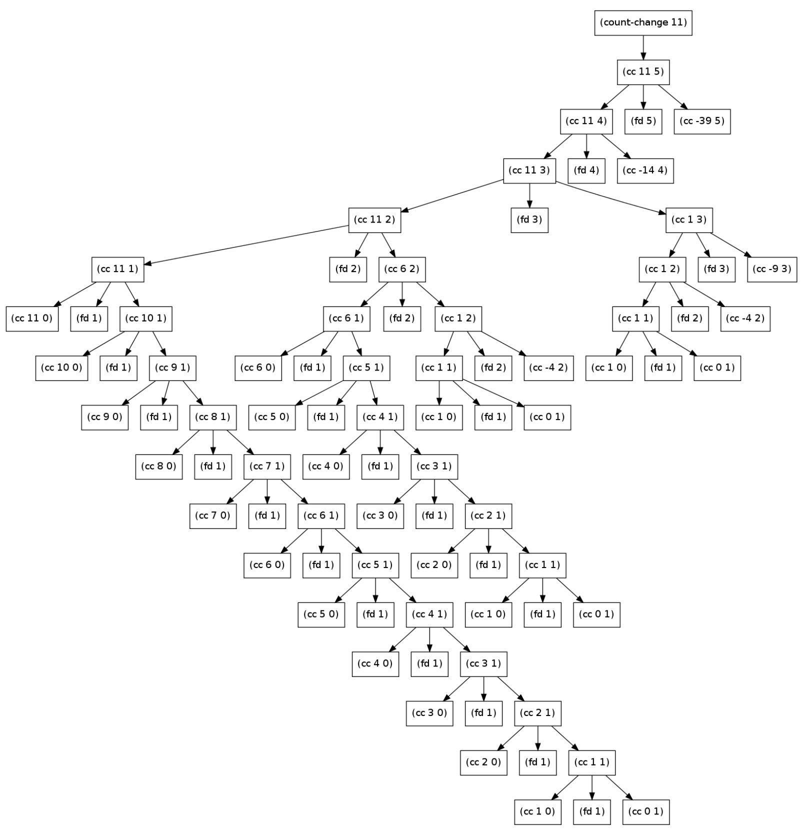
   Draw the tree illustrating the process generated by the `count-change` procedure of section *Note 1-2-2:: in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

2. Answer

   First, we'll setup some tooling to output to GraphViz's `dot` format. This could be done in more complex (and interesting) ways, but this tries to stick as closely as possible to the Scheme features that have been discussed in the book so far. The additional features used are `format` and `printf` (for displaying output) and `random` for creating a sort-of-unique ID. If we were to just use the information available in a procedure (that is, its name and the parameters with which it was called), we wouldn't have a tree, as multiple calls to the same procedure with the same parameters would be collapsed.

```
(define (random-id)
  (random 5000000))

(define (make-name str)
  ;; Append random number to given string for a hopefully unique node
  ;; name.  This isn't perfect, as there is a small possibility that
  ;; IDs could be repeated.  As we aren't using assignment yet, this
  ;; is probably good enough.
  (format "~a_~a" str (random-id)))

(define (dot-node name label)
  (printf "    ~a [shape=box,label=\"~a\"];~n" name label))

(define (dot-edge parent child)
  (printf "    ~a -> ~a;~n" parent child))

(define (count-change amount)
  (define name (make-name "count_change"))
  (dot-node name (format "(count-change ~a)" amount))
  (cc amount 5 name))

(define (cc amount kinds-of-coins parent)
  (define name (make-name "cc"))
  (dot-node name (format "(cc ~a ~a)" amount kinds-of-coins))
  (dot-edge parent name)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1)
                     name)
                 (cc (- amount
                        (first-denomination kinds-of-coins name))
                     kinds-of-coins
                     name)))))
```

```
(define (first-denomination kinds-of-coins parent)
  (define name (make-name "fd"))
  (dot-node name (format "(fd ~a)" kinds-of-coins))
  (dot-edge parent name)

  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))

(count-change 11)
```



1.3.2 Exercise 1.15: Approxmiating sine

The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x \approx x$ if x is sufficiently small, and the trigonometric identity

$$\sin x = 3\sin\frac{x}{3} - 4\sin^3\frac{x}{3}$$

to reduce the size of the argument of *sin*. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:

```
(define (cube x) (* x x x))

(define (p x)
  ;; (Modified to show calls to p)
  (printf "(p ~a)~n" x)
  (- (* 3 x) (* 4 (cube x))))

(define (sine angle)
   (if (not (> (abs angle) 0.1))
       angle
       (p (sine (/ angle 3.0)))))
```

a. How many times is the procedure `p` applied when `(sine 12.15)` is evaluated?

```
(sine 12.15)
```

5 calls to `p`.

b. What is the order of growth in space and number of steps (as a function of a) used by the process generated by the `sine` procedure when `(sine a)` is evaluated?

Logarithms answer, more or less, the question "how many times can I divide one number by another". The second number is the *base*. So, consider log base 2 of 8: 8/2=4, 4/2=2, 2/2=1; thus, $log_2 8 = 3$.

The actual definition is that the log of a number is the exponent to which the base must be raised to equal that number. Thus, since $2^3 = 8$, then $log_2 8 = 3$.

As can be seen by the single call to `p` in the body of `sine`, each recursive call reduces `angle` by a factor of 3; thus, `p` is going to be of $\theta(\log n)$. (The specific logarithmic base is effectively a constant, so all logarithmic processes are considered to be of them same order of computational complexity.)

1.4 Section 1.2.4 Exponentiation

1.4.1 Exercise 1.16: Iterative exponentiation in logarithmic time

1. Problem

   Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast-expt`. (Hint: Using the observation that $(b^{(}n/2))^2 = (b^2)^{(}n/2)$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an "invariant quantity" that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

2. Answer

   Here's the code from the section for the original `fast-expt` algorithm.

   ```
   (define (square n)
     (* n n))

   (define (even? n)
     (= (remainder n 2) 0))

   (define (fast-expt b n)
     (cond ((= n 0) 1)
           ((even? n) (square (fast-expt b (/ n 2))))
           (else (* b (fast-expt b (- n 1))))))
   ```

   Note that the `even?` case in the `cond` is building up calls to `square` and the `else` is building up calls to `*`. The stack of calls to `fast-expt` keeps building up until it bottoms out with the first case, after which all of the pending computations can be rolled back up.

   This problem is to switch from this from a logarithmic recursive process to a logarithmic iterative process.

   ```
   (define (fast-expt-2 b n)
     (define (fast-expt-iter b n a)
       (cond ((= n 0) a)
   ```

```
          ((even? n) (fast-expt-iter (square b) (/ n 2) a))
          (else (fast-expt-iter b (- n 1) (* a b)))))
  (fast-expt-iter b n 1))
```

To evaulate this, we'll reuse the comparison function used for Problem 1.11:

```
(define (compare f1 f2 k)

  (define (compare-iter i good?)
    (define t0 (current-milliseconds))
    (define r1 (f1 i))
    (define t1 (current-milliseconds))
    (define r2 (f2 i))
    (define t2 (current-milliseconds))

    (printf "~a: ~a (~a ms) ~a (~a ms) => ~a~n"
            i
            r1 (- t1 t0)
            r2 (- t2 t1)
            (= r1 r2))
    (if (< i k)
        (compare-iter (+ i 1) (and good? (= r1 r2)))
        (and good? (= r1 r2))))

  (compare-iter 1 true))

(define (expt-by-two n) (fast-expt 2 n))
(define (expt-by-two-2 n) (fast-expt-2 2 n))
(compare expt-by-two expt-by-two-2 20)
```

1.4.2 Exercise 1.17: Recursive integer multiplication with `square` and `halve`

1. Problem

   The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the `expt` procedure:

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

This algorithm takes a number of steps that is linear in b. Now suppose we include, together with addition, operations `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to 'fast-expt' that uses a logarithmic number of steps.

2. Answer This is a straightforward translation of the `fast-expt` code from the text to the multiplication problem... the problem (and solution) have exactly the same shape.

```
(define (double n) (* n 2))

(define (halve n) (/ n 2))

(define (even? n)
  (= (remainder n 2) 0))

(define (fast-mult a b)
  (cond ((= b 0) 0)
        ((even? b) (double (fast-mult a (halve b))))
        (else (+ a (fast-mult a (- b 1))))))
```

1.4.3 Exercise 1.18: Iterative integer multiplication with `square` and `halve`

1. Problem

   Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

2. Answer

   This solution is also straightforward. The only trick part is keeping straight what needs to be added and subtracted, and from where.

```
(define (fast-mult-2 a b)
  (define (fast-mult-iter a b c)
    (cond ((= b 0) c)
          ((even? b) (fast-mult-iter (double a) (halve b) c))
          (else (fast-mult-iter a (- b 1) (+ c a)))))
  (fast-mult-iter a b 0))
```

1.4.4 WRITEUP Exercise 1.19: Logarithmic Fibonacci calculations

1. Problem

   There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state variables `a` and `b` in the `fib-iter` process of section 1.2.2: $a \leftarrow a+b$ and $b \leftarrow a$. Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair Fib$(n+1)$ and Fib(n). In other words, the Fibonacci numbers are produced by applying T^n, the nth power of the transformation T, starting with the pair $(1,0)$. Now consider T to be the special case of $p=0$ and $q=1$ in a family of transformations T_{pq}, where T_{pq} transforms the pair (a,b) according to $a \leftarrow bq+aq+ap$ and $b \leftarrow bp+aq$. Show that if we apply such a transformation T_{pq} twice, the effect is the same as using a single transformation $T_{p'q'}$ of the same form, and compute p' and q' in terms of p and q. This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring, as in the `fast-expt` procedure. Put this all together to complete the following procedure, which runs in a logarithmic number of steps:

```
(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   <??>      ; compute p'
                   <??>      ; compute q'
                   (/ count 2)))
```

```
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

2. Answer

```
(define (fib-t n)
  (fib-iter 1 0 0 1 n))

(define (fib-t-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-t-iter a
                     b
                     (+ (square q) (square p))
                     (+ (* 2 p q) (square q))
                     (/ count 2)))
        (else (fib-t-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

1.5 Section 1.2.5 Greatest Common Divisors

1.5.1 TODO Exercise 1.20: Revisiting applicative order and normal order

1. Problem The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative `gcd` procedure given above. Suppose we were to interpret this procedure using normal-order evaluation, as discussed in section 1.1.5. (The normal-order-evaluation rule for `if` is described in Exercise 1.5.) Using the substitution method (for normal order), illustrate the process generated in evaluating `(gcd 206 40)` and indicate the `remainder` operations that are actually performed. How many `remainder` operations are actually performed in the normal-order evaluation of `(gcd 206 40)`? In the applicative-order evaluation?

2. Answer

   Recall that for applicative order, arguments are first evaluated, then the procedure is applied. For normal order, everything is fully expanded before the arguments are evaluated.

1.6 Section 1.2.6 Testing for Primality

1.6.1 Exercise 1.21: Using `smallest-divisor`

1. Problem Use the 'smallest-divisor' procedure to find the smallest divisor of each of the following numbers: 199, 1999,

   (a)

2. Answer

   First, the relevant code from section 1.2.6:

```
(require (planet neil/sicp:1:17))

(define (square n) (* n n))

(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))

(smallest-divisor 199)

(smallest-divisor 1999)

(smallest-divisor 19999)
```

1.6.2 Exercise 1.22: Measuring runtime

1. Problem Most Lisp implementations include a primitive called 'runtime' that returns an integer that specifies the amount of time the system has been running (measured, for example, in microseconds). The following 'timed-prime-test' procedure, when called with an integer n, prints n and checks to see if n is prime. If n is prime, the procedure prints three asterisks followed by the amount of time used in performing the test.

   ```
   ;; This code has been tweaked slightly to return true/false so
   ;; the return value can be used in tests. Also, only displays
   ;; output for prime numbers.

   (define (timed-prime-test n)
     ;(display n)
     ;(display " ")
     (start-prime-test n (runtime)))

   (define (start-prime-test n start-time)
     (if (prime? n)
         (report-prime n (- (runtime) start-time))
         false))

   (define (report-prime prime elapsed-time)
     (display prime)
     (display " *** ")
     (display elapsed-time)
     (newline)
     true)
   ```

   Using this procedure, write a procedure 'search-for-primes' that checks the primality of consecutive odd integers in a specified range. Use your procedure to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth of [theta]($_{\text{[sqrt](n)}}$), you should expect that testing for primes around 10,000 should take about _[sqrt]$_{(10)}$ times as long as testing for primes around 1000. Do your timing data bear this out? How well do the data for 100,000 and 1,000,000 support the _[sqrt]$_{(n)}$

prediction? Is your result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation?

2. Answer

   First, a procedure to scan a range of consecutive odd numbers for primality:

```
(define (odd? n) (= (remainder n 2) 1))

;; Find primes in range from a to b
(define (search-for-primes a b)
  (if (< a b)
      (cond ((odd? a)
             (timed-prime-test a)
             (search-for-primes (+ a 2) b))
            (else
             (search-for-primes (+ a 1) b)))))

;; Find the first k primes larger than n
(define (find-k-primes k n)
  (if (odd? n)
   (if (> k 0)
       (if (timed-prime-test n)
           (find-k-primes (- k 1) (+ n 2))
           (find-k-primes k (+ n 2))))
   (find-k-primes k (+ n 1))))

;; Starting with =, find the first k higher primes;
;; then multiply n by 10 and repeat intervals times.
(define (prime-scan k intervals n)
  (find-k-primes k n)
  (if (> intervals 1) (prime-scan k (- intervals 1) (* n 10))))
```

   Using this, finding the first three primes larger than 1,000 is easy: 1,009, 1,013, and 1,019.

```
(find-k-primes 3 1000)
```

   And for 10,000, 100,000, and 1,000,000::

```
(prime-scan 3 3 (expt 10 4))
```

As can be seen, on my machine, calculating primality using this method for numbers around 10^4 takes about 8-9 microseconds, 10^5 takes about 24-25 microseconds, and 10^6 takes around 77-78 microseconds.

```
(display (* 8 (sqrt 10)))
(newline)
(display (* 25 (sqrt 10)))
```

These results match almost perfectly to the predicted execution time. The tweaked version of `prime-scan` makes it easy to test this at a broad range of magnitudes:

```
(prime-scan 1 13 (expt 10 4))
```

For a quick-and-dirty evaluation of this output, we'll munge it fast in the shell.

```
# Separate code block here so we can reuse these results easily in the
# next exercise
cut -f3 -d' '
```

```
echo Measured Predicted
for t in $(cut -f3 -d' '); do
    if [ -n "${last}" ]; then
        # dc is an ancient RPN calculator
        # space pushes a number, 'v' is sqrt
        # and 'p' prints the value on the top of the stack
        guess=$(dc -e"${last} 10v*p")
    fi
    echo $t $guess
    last=$t
done
```

These results continue to stay close to the predicted values which supports the analysis that run time is proportional to the number of steps required for the computation.

1.6.3 Exercise 1.23: Speeding up `smallest-divisor`

1. Problem The 'smallest-divisor' procedure shown at the start of this section does lots of needless testing: After it checks to see if the number is divisible by 2 there is no point in checking to see if it is divisible by any larger even numbers. This suggests that the values used for 'test-divisor' should not be 2, 3, 4, 5, 6, ..., but rather 2, 3, 5, 7, 9, ... To implement this change, define a procedure 'next' that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the 'smallest-divisor' procedure to use '(next test-divisor)' instead of '(+ test-divisor 1)'. With 'timed-prime-test' incorporating this modified version of 'smallest-divisor', run the test for each of the 12 primes found in *Note Exercise 1-22::. Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

2. Answer

   This is a little messy since we haven't had higher-order functions introduced, yet, so here are all of the relevant functions rewritten to use the new `better-smallest-divisor` procedure.

   ```
   (define (next test-divisor)
     (if (= test-divisor 2)
         3
         (+ test-divisor 2)))

   (define (better-smallest-divisor n)
     (better-find-divisor n 2))

   (define (better-find-divisor n test-divisor)
     (cond ((> (square test-divisor) n) n)
           ((divides? test-divisor n) test-divisor)
           (else (better-find-divisor n (next test-divisor)))))

   (define (better-prime? n)
     (= n (better-smallest-divisor n)))

   (define (better-timed-prime-test n)
   ```

```
  ;(display n)
  ;(display " ")
  (better-start-prime-test n (runtime)))

(define (better-start-prime-test n start-time)
  (if (better-prime? n)
      (report-prime n (- (runtime) start-time))
      false))

(define (better-find-k-primes k n)
  (if (odd? n)
   (if (> k 0)
       (if (better-timed-prime-test n)
           (better-find-k-primes (- k 1) (+ n 2))
           (better-find-k-primes k (+ n 2))))
   (better-find-k-primes k (+ n 1))))

;; Starting with =, find the first k higher primes;
;; then multiply n by 10 and repeat intervals times.

(define (better-prime-scan k intervals n)
  (better-find-k-primes k n)
  (if (> intervals 1) (better-prime-scan k (- intervals 1) (* n 10))))
```

Here are the 12 primes that are specified in the exercise:

```
(better-prime-scan 3 3 (expt 10 4))
```

These results are very close to those for the original version... but, since these magnitudes are quite small relative to numbers that would have been expensive to calculate in 1996 (when SICP 2ed was published), it's difficult to differentiate. More useful is comparing at larger magnitudes:

```
(better-prime-scan 1 13 (expt 10 4))
```

```
cut -f3 -d' '
```

```
(mapcar* 'append first second)
```

First column is the original figures, second is the `better-*` version…it's a bit fussy to get the headers added into an org-babel block that combines two sets of output:

5	5
14	15
44	47
138	148
440	460
1396	1465
4488	5174
16210	17281
45320	52689
141999	177798
452743	481867
1411103	870297
4399915	2771143

1.6.4 WRITEUP Exercise 1.24: Putting the Fermat method to work

1. Problem

   Modify the `timed-prime-test` procedure of Exercise 1.22 to use `fast-prime?` (the Fermat method), and test each of the 12 primes you found in that exercise. Since the Fermat test has $\theta \log n$) growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Do your data bear this out? Can you explain any discrepancy you find?

2. Answer

   First, the code from Section 1.2.6. Since we're pushing the input size larger than 4294967087, we can't use Racket's built-in `random`, so an external library from Planet (Racket's package repository) is used that does not cap the range (`williams/science/random-source`).

```
(require (planet williams/science/random-source))

(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
```

```
         (remainder (square (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m))))

(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random-integer (- n 1)))))

(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

Now, we need to modify the relevant procedures to use `fast-prime?`.

```
(define (fast-timed-prime-test n times)
  (fast-start-prime-test n times (runtime)))

(define (fast-start-prime-test n times start-time)
  (if (fast-prime? n times)
      (report-prime n (- (runtime) start-time))
      false))

(define (fast-find-k-primes k n times)
  (if (odd? n)
   (if (> k 0)
       (if (fast-timed-prime-test n times)
           (fast-find-k-primes (- k 1) (+ n 2) times)
           (fast-find-k-primes k (+ n 2) times)))
   (fast-find-k-primes k (+ n 1) times)))

(define (fast-prime-scan k intervals n times)
  (fast-find-k-primes k n times)
  (if (> intervals 1) (fast-prime-scan k (- intervals 1) (* n 10) times)))
```

Using only 10 tests is super-fast:

```
(fast-prime-scan 1 13 (expt 10 4) 10)
```

Compare this to 100:

```
(fast-prime-scan 1 13 (expt 10 4) 100)
```

And to 1,000:

```
(fast-prime-scan 1 13 (expt 10 4) 1000)
```

Increasing the number of `times` that the `fast-prime?` test is performed linearly increases the runtime. But, overall, it can be seen that the rate of growth is **much** slower than the original `prime?` and `better-prime?` procedures. Making this more concrete:

```
# Separate code block here so we can reuse these results easily in the
# next exercise
cut -f3 -d' '
```

```
echo Measured Predicted
for t in $(cut -f3 -d' '); do
    if [ -n "${last}" ]; then
        # annoyingly, dc doesn't have a log function, so this uses the
        # completely non-standard (but readily available) qalc package.
        guess=$(qalc -set "approximation 2" -t "log(2.9**(${last}) * 2.9,2.9)")
    fi
    echo $t $guess
    last=$t
done
```

1.6.5 Exercise 1.25: A not-so-fast use of `fast-expt`

1. Problem Alyssa P. Hacker complains that we went to a lot of extra work in writing `expmod`. After all, she says, since we already know how to compute exponentials, we could have simply written

```
(define (bad-expmod base exp m)
  (remainder (fast-expt base exp) m))
```

Is she correct? Would this procedure serve as well for our fast prime tester? Explain.

2. Answer

First, recall the relevant supporting code:

```
(define (square n)
  (* n n))

(define (even? n)
  (= (remainder n 2) 0))

(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

And compare the problem's definition of `expmod` with the one used for Section 1.24:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m))))
```

The final result of both the original `expmod` and Alyssa's `bad-expmod` will be the same: they both calculate $base^{exp} \mod m$. `bad-expmod` has to do a lot more work to achieve the same end, though, as it's manipulating much longer numbers: it generates the full exponential value before applying `remainder`… and `remainder` gets put through the wringer as it must divide that very large number by `m`. The original `expmod`, on the other hand, applies `remainder` at every step of the way, keeping the number in the range where it is both useful and easier to manipulate.

To test this, we'll set up a batch of procedures to use `bad-expmod`.

```
(define (bad-fermat-test n)
  (define (try-it a)
    (= (bad-expmod a n n) a))
  (try-it (+ 1 (random-integer (- n 1)))))

(define (bad-prime? n times)
  (cond ((= times 0) true)
        ((bad-fermat-test n) (bad-prime? n (- times 1)))
        (else false)))

(define (bad-timed-prime-test n times)
  (bad-start-prime-test n times (runtime)))

(define (bad-start-prime-test n times start-time)
  (if (bad-prime? n times)
      (report-prime n (- (runtime) start-time))
      false))

(define (bad-find-k-primes k n times)
  (if (odd? n)
   (if (> k 0)
       (if (bad-timed-prime-test n times)
           (bad-find-k-primes (- k 1) (+ n 2) times)
           (bad-find-k-primes k (+ n 2) times)))
   (bad-find-k-primes k (+ n 1) times)))

(define (bad-prime-scan k intervals n times)
  (bad-find-k-primes k n times)
  (if (> intervals 1) (bad-prime-scan k (- intervals 1) (* n 10) times)))
```

Here are some values for `fast-prime?` using the original `expmod`, using a small number of tests (just 10):

```
(fast-prime-scan 1 4 100 10)
```

Barely any time at all, on the order of 10 microseconds. The new `bad-expmod` approach, however, does indeed live up to the name:

```
(bad-prime-scan 1 4 100 10)
```

For even an input as small as 10^5, the runtime is already nearing a second! This exercise is a great demonstration of potentially difficult to notice computation complexity bottlenecks, and the importance of picking the right algorithm for the job.

1.6.6 Exercise 1.26: A subtle slowdown in `expmod`

Louis Reasoner is having great difficulty doing Exercise 1-24. His `fast-prime?` test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the `expmod` procedure to use an explicit multiplication, rather than calling `square`:

```
(define (slow-expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (slow-expmod base (/ exp 2) m)
                       (slow-expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (slow-expmod base (- exp 1) m))
                    m))))
```

"I don't see what difference that could make," says Louis. "I do." says Eva. "By writing the procedure like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process." Explain.

1. Answer

   Once again, recall the original `expmod` procedure:

   ```
   (define (expmod base exp m)
     (cond ((= exp 0) 1)
           ((even? exp)
            (remainder (square (expmod base (/ exp 2) m))
                       m))
           (else
            (remainder (* base (expmod base (- exp 1) m))
                       m))))
   ```

This is a lovely and subtle change. While appearing to be a simple in-place substitution of a procedure, it actually changes the single recursive call to `expmod` to be a tree of recursive calls, with two recursive calls at each internal node of the tree.

The original version divides the size of n by two at each stage... since n can only be divided by 2 at most $\log_2 n$ times, this gives the expected complexity. While `slow-expmod` also divides the size of its argument by two, it also generates two recursive calls, one for each half. Thus, is does not reduce the size of the overall problem to be solved: while the tree has only $log_2 n$ levels, there are 2^k subproblems at each level k. (Level 0 has a single problem; level 1 has $2^1 = 2$ problem. Each of those two problems generates two recursive children for $2^2 = 4$ problems at level two, and so on.

So, given $\Theta(\log_2 2^n)$, the log and the exponential cancel each other out (by the definition of logarithm, and the overall complexity is $Θ(n).

1.6.7 Exercise 1.27: Fooling Fermat with Carmichael numbers

1. Problem

   Demonstrate that the Carmichael numbers listed in *Note Footnote 1-47:: really do fool the Fermat test. That is, write a procedure that takes an integer n and tests whether a^n is congruent to $a \mod n$ for every $a < n$, and try your procedure on the given Carmichael numbers.

2. Answer

```
(define (verify-fermat n)
  (define (verify-fermat-iter a n)
    (cond ((>= a n)
           true)
          ((= (expmod a n n) a)
           (verify-fermat-iter (+ a 1) n))
          (else
           false)))
  (verify-fermat-iter 1 n))

(define (descriptive-verify-fermat n)
  (display n)
  (if (verify-fermat n)
```

```
      (if (prime? n)
          (display ": prime and correctly passes the Fermat test")
          (display ": not prime and incorrectly passes the Fermat test"))
      (if (prime? n)
          (display ": prime and incorrectly fails the Fermat test")
          (display ": not prime and correctly fails the Fermat test")))
  (newline))

(define (fermat-scan-range a b)
  (descriptive-verify-fermat a)
  (if (< a b)
      (fermat-scan-range (+ a 1) b)))
```

This procedure does indeed show that the first six Carmichael numbers slip through the Fermat test.

```
(descriptive-verify-fermat 561)
(descriptive-verify-fermat 1105)
(descriptive-verify-fermat 1729)
(descriptive-verify-fermat 2465)
(descriptive-verify-fermat 2821)
(descriptive-verify-fermat 6601)

(fermat-scan-range 1101 1109)
```

For a bit more fun, we can turn this into a test for Carmichael numbers and find them ourselves. This could be much more fun with lists, `map`, and `filter`, but we haven't had them introduced, yet, so this sticks with printing out the relevant numbers.

```
(define (carmichael? n)
   (and (not (prime? n)) (verify-fermat n)))

(define (carmichael-scan-range a b)
  (if (carmichael? a) (printf "~a~n" a))
  (if (< a b) (carmichael-scan-range (+ a 1) b)))
```

Here's an example of using this to find all the Carmichael numbers under 10,000. As can be seen, the first six numbers mentioned in the text are all found using this method.

```
(carmichael-scan-range 1 100000)
```

1.6.8 WRITEUP Exercise 1.28: The Miller-Rabin test

1. Problem

   One variant of the Fermat test that cannot be fooled is called the "Miller-Rabin test" (Miller 1976; Rabin 1980). This starts from an alternate form of Fermat's Little Theorem, which states that if n is a prime number and a is any positive integer less than n, then a raised to the $(n - 1)$st power is congruent to 1 modulo n. To test the primality of a number n by the Miller-Rabin test, we pick a random number $a < n$ and raise a to the $(n - 1)$st power modulo n using the `expmod` procedure. However, whenever we perform the squaring step in `expmod`, we check to see if we have discovered a "nontrivial square root of 1 modulo n," that is, a number not equal to 1 or $n-1$ whose square is equal to 1 modulo n. It is possible to prove that if such a nontrivial square root of 1 exists, then n is not prime. It is also possible to prove that if n is an odd number that is not prime, then, for at least half the numbers $a < n$, computing $a^(n-1)$ in this way will reveal a nontrivial square root of 1 modulo n. (This is why the Miller-Rabin test cannot be fooled.) Modify the `expmod` procedure to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller-Rabin test with a procedure analogous to `fermat-test`. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make `expmod` signal is to have it return 0.

2. Answer

   This has some ugly bits... judicious use of `let` (which isn't introduced until the next section) would again simplify some of these expressions.

```
;; Test whether i is a nontrivial square root of 1 modulo m
(define (nontrivial-sqrt-mod? i m)
  (and (not (= i 1))
       (not (= i (- m 1)))
       (= (remainder (square i) m) 1)))

(define (mr-expmod base exp m)
  (define (maybe-continue i)
    (if (or (= i 0) (nontrivial-sqrt-mod? i m))
        0
        (remainder (square i) m)))
```

```
  (cond ((= exp 0) 1)
        ((even? exp)
         (maybe-continue (mr-expmod base (/ exp 2) m)))
        (else
         (remainder (* base (mr-expmod base (- exp 1) m))
                    m))))

(define (mr-test n)
  (define (try-it a)
    ;; We don't need to check if the return of mr-expmod = 0,
    ;; as it is always the case that a>1.
    (= (mr-expmod a (- n 1) n) 1))
  (try-it (+ 1 (random-integer (- n 1)))))

(define (mr-prime? n times)
  ;; We have to special-case n=1 and n=2.
  ;; (Note prime? incorrectly reports 1 as prime, and
  ;; fast-prime also fails outright.)
  (cond ((= times 0) true)
        ((= n 1) false)
        ((= n 2) true)
        ((mr-test n) (mr-prime? n (- times 1)))
        (else false)))

(define (mr-timed-prime-test n times)
  (mr-start-prime-test n times (runtime)))

(define (mr-start-prime-test n times start-time)
  (if (mr-prime? n times)
      (report-prime n (- (runtime) start-time))
      false))

(define (mr-find-k-primes k n times)
  (if (odd? n)
      (if (> k 0)
          (if (mr-timed-prime-test n times)
              (mr-find-k-primes (- k 1) (+ n 2) times)
              (mr-find-k-primes k (+ n 2) times)))
      (mr-find-k-primes k (+ n 1) times)))
```

```
(define (mr-prime-scan k intervals n times)
  (mr-find-k-primes k n times)
  (if (> intervals 1) (mr-prime-scan k (- intervals 1) (* n 10) times)))
```