Assignment 6 Solutions

Nathan Hitchings

January 31, 2019

1 Exercise 1.21

Use the `smallest-divisor` procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

1.1 Solution

This is the `smallest-divisor` procedure:

```
#lang sicp

(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 (square x) (* x x))
```

Applying `smallest-divisor` to each of 199, 1999, and 19999 yields

```
(smallest-divisor 199)   ; = 199
(smallest-divisor 1999)  ; = 1999
(smallest-divisor 19999) ; = 7
```

2 Exercise 1.22

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.

```
#lang sicp

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

(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))

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

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

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 $\theta(\sqrt{n})$, 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 $\theta(\sqrt{n})$ 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?

2.1 Solution

```
(define (search-for-primes start end)
(cond ((> start end) (display "Done\n"))
      ((even? start) (search-for-primes (+ start 1) end))
      (else (begin
              (timed-prime-test startp)
              (search-for-primes (+ start 1) end)))))
```

prime number	time (in microseconds?)
1009	1
1013	1
1019	1
10007	2
10009	1
10037	1
100003	5
100019	5
100043	5
1000003	15
1000033	15
1000037	15
10000019	48
10000079	48
10000103	47

3 Exercise 1.23

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 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?

3.1 Solution

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

(define (find-divisor n test-divisor)
  (define (next x)
    (cond [(= 2 x) 3]
          [else (+ x 2)]))
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (next test-divisor)))))
```

This is the output from running this version of the program on the primes in thne previous exercise:

prime	time
1009	5
1013	1
1019	1
10007	4
10009	4
10037	4
100003	6
100019	23
100043	6
1000003	16
1000033	16
1000037	16
10000019	61
10000079	50
10000103	50

4 Exercise 1.24

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?

4.1 Solution

prime	time
1009	276
1013	4
1019	5
10007	5
10009	5
10037	5
100003	16
100019	6
100043	5
1000003	6
1000033	6
1000037	6
10000019	7
10000079	8
10000103	7

5 Exercise 1.25

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 (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.

6 Exercise 1.26

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 (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (expmod base (/ exp 2) m)
                       (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base
                       (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.

6.1 Solution

The reason that the process is now $\theta(n)$ is that the expression `(expmod base (/ exp 2) m)` must now be evaluated twice for every call while using `square` allows for the expression to be evaluted only once.

7 Exercise 1.27

Demonstrate that the Carmichael numbers listed in 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 modulo n for every $a < n$, and try your procedure on the given Carmichael numbers.

7.1 Solution

```
(define (square x) (* x x))
(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 (test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (define (loop a)
    (cond [(= a 0) true]
          [(try-it a) (loop (- a 1))]
          [else false]))
  (loop n))
```

8 Exercise 1.28

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.

8.1 Solution

```
(define (square x) (* x x))
(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)]))
```