MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1985

Problem set 2

Issued:	Tuesday, February 12
Due:	in recitation on Friday, February 22 *for all sections*
Reading:	Text, Chapter 1, Sections 1.2 and 1.3

Homework exercises

Write up and turn in the following exercises from the text:

- Exercise 1.25: Products
- Exercise 1.26: Accumulate
- Exercise 1.26: Filters
- Exercise 1.28: `(f f)`
- Exercise 1.32: Repeated application
- Exercise 1.33: Repeated smoothing

Laboratory Assignment: Testing for Primality

This laboratory assignment deals with the orders of growth in the number of steps required by some algorithms that test whether a given number is a prime. It is based upon Section 1.2.6 of the text. You should read that material before beginning work on this assignment.

To Do at the Laboratory

All the procedures from section 1.2.6 are installed on the Chipmunk shared resource manager, and can be loaded onto your floppy disk. To do this, follow the instructions given in the Chipmunk manual in the section on "Loading Problem Set Files," to load the code for problem set 2. (The code for this problem set is sufficiently short that we have not separated out the procedures that you are expected to modify, as explained in the Chipmunk manual.)

There are slight differences between the `timed-prime-test` and `fermat-test` procedures we have given you and the ones in the book. `timed-prime-test` uses `princ` instead of `print` to avoid starting a new line until it gets to the next number being tested. `fermat-test` uses `big-random` instead of `random` because we will be testing some very big numbers for primality, and the `random` procedure built in to Scheme cannot generate extremely large random numbers. `big-random` is defined as

```
(define (big-random n)
  (random (min n (expt 10 10))))
```

This procedure can be called with arbitrarily large numbers, but will only generate numbers less than or equal to 10^{10}.

Do exercise 1.16 from the text.

Define the following procedure, which, when called with an odd integer n, tests the primality of consecutive odd integers starting with n. (Obviously, there is no point checking if even integers are prime.) The procedure will keep running until you interrupt it by typing ctrl-G.

```
(define (search-for-primes n)
  (timed-prime-test n)
  (search-for-primes (+ n 2)))
```

Transfer your new procedure to Scheme and test it. When you are satisfied that it is working, use `search-for-primes` to find the 3 smallest primes that are larger than larger than 1,000; larger than 10,000; larger than 100,000; larger than 1,000,000.

Prepare a chart as follows:

Range: 1000

Primes:	prime1	prime2	prime3
Time1:			
Time2:			
Time3:			

and make similar charts for the ranges 10,000, 100,000, and 1,000,000. In the spaces marked prime$n$, fill in the three smallest primes you found. (You will fill in 12 primes in all, 3 in the chart for each range.) Under each prime, in the row labelled Time1, fill in the time used by the prime testing procedure to determine that the number was prime. (The Scheme primitive procedure `runtime` used in `timed-prime-test` returns the amount of time in hundredths of a second that the system has been running.)[1]

Modify `smallest-divisor` as described in exercise 1.18 in the text. With `timed-prime-test` using this modified version of `smallest-divisor`, run the test for each of the 12 primes listed in your table. Enter the times required by the tests in your chart in the rows labelled Time2.

Back in the editor, modify `timed-prime-test` to use `fast-prime?` in place of `prime?`. You can assume that a number is prime if it passes the Fermat test two times. Now test each of your 12 primes as in Part 3 above and enter the required times in the spaces marked Time3.

Now let's test some really big numbers for primality. In 1644, the French mathematician Marin Mersenne published the claim that numbers of the form $2^p - 1$ are prime for the following values of p and for no other p less than 257.

$$2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$$

[1]Be careful here. `runtime` counts not only actual compute time, but also garbage collection time. Garbage collection is a process that Scheme goes through as part of its memory management. (We will discuss this near the end of the term.) When Scheme is garbage-collecting, the letter G appears at the lower right-hand corner of the screen. (Ordinarily, the Greek letter pi is shown there.) If a garbage collection happens during one of the computations whose time you care about, the `runtime` number will be too large. If you see a garbage collection happening during one of the computations for a number in your chart, it's best to retime the computation for that number.

It turns out that Mersenne missed a few values of p, and that some of the values in his list do not give primes. Use `timed-prime-test` (modified as in Part 4 to use `fast-prime?`) to determine which of the values in Mersenne's list give primes, and which do not. (Prime numbers of the form $2^p - 1$ are known as *Mersenne primes.*) Record the time required for each test. To aid in testing, it will help to define the following procedure:

```
(define (mersenne p) (- (expt 2 p) 1))
```

Let's try to find which values of p Mersenne missed. A straightforward way to do this is to write an iterative procedure that checks, for all integers p in a given range, whether `(mersenne` p`)` is prime. If we blindly test all integers in a given range, however, then we will be doing a lot of needless checking, because $2^p - 1$ cannot be prime unless p is prime. (Sketch of proof: Show that $2^{ab} - 1$ is divisible by $2^a - 1$.) Write a procedure `mersenne-range` that works as follows: For each p in a given range of integers, the program should first use `prime?` to test whether p is prime.[2] If so, it should use `fast-prime?` to test whether `(mersenne` p`)` is prime. As the program runs, it should print p, `(mersenne` p`)`, and the result of the prime test. Use `mersenne-range` to find all values of p less than or equal to 257 for which $2^p - 1$ is prime.

Generalize `mersenne-range` to a procedure called `prime-filter-check`, which takes as input two integers `a` and `b`, a predicate called `filter`, and a procedure of one argument called `term`. For each integer n in the range from `a` to `b` for which `(filter` n`)` is true, the program should check (using `fast-prime?`) whether `(term` n`)` is prime. As a test, calling the procedure with `filter` as `prime?` and `term` as `mersenne` should do the same thing as the `mersenne-range` procedure of Part 6.

It is a curious fact that numbers of the form $n^2 + n + 41$ are prime for small values of n. Use the procedure you wrote in Part 7 to answer the following questions:

1. What are the ten smallest positive integers n for which $n^2 + n + 41$ is not prime?
2. What are the ten smallest *primes* p for which $n^2 + n + 41$ is not prime?

Post-lab Write-up

After you are through at the lab, you are to write up and hand in answers to the following questions:

1. What is the smallest divisor of each of the numbers that you tested in Part 1?
2. Prepare a neat copy of the table you made, listing the 3 primes in each of the 4 ranges and the corresponding timings for each of the three primality tests.
3. The number of steps in the first prime test should grow as fast as the square root of the number being tested. We should expect therefore that testing for primes around 10,000 should take about $\sqrt{10}$ times as long as testing for primes around 1000. Does your timing data for Time1 bear this out? How well does the data for 100,000 and 1,000,000 support the $\sqrt{n}$ prediction?
4. The modification you made in Part 3 was designed to speed up the prime test by halving the number of test steps. So you should expect it to run about twice as fast. Does your data

[2] If you like, you can use `fast-prime?` here. But the numbers p will be rather small so there is not much advantage. Also, `fast-prime?` with only a couple of iterations is unreliable for very small numbers. Also be careful with $p = 2$, for which `fast-prime?` does not work.

bear this out? For each of the primes in your table, compute the ratio Time1/Time2. Does this ratio appear to be constant (i.e., more or less independent of the number being tested)? Does the ratio indicate that the new test runs twice as fast as the old? If not, what is the actual ratio and how do you explain the fact that it is different from 2?

5. The number of steps required by the `fast-prime?` test has $O(\log n)$ growth. How then would you expect the time to test primes near 100,000 to compare with the time needed to test primes near 1000? How well does your data bear this out? Can you explain any discrepancy you find?

6. Which of the numbers in Mersenne's list do in fact yield primes? What numbers did Mersenne miss?

7. In Part 5, how long did your program take to test primality of $2^{127} - 1$? Estimate how long the first prime testing algorithm (Part 1) would require to check the primality of this number? To answer this question, extrapolate from the data you collected in the Time1 entries of your table, together with the fact that the number of steps required grows as $\sqrt{n}$. Give the answer, not in seconds, but in whatever unit seems to most appropriately express the amount of time required (e.g., minutes, hours, days, ...). Be sure to explain how you arrived at your estimate.

8. Turn in listings of the `mersenne-range` and `prime-filter-check` procedures that you wrote in Parts 6 and 7.

9. What are the answers you obtained in Part 8? How did you use the procedure in Part 7 to find these answers?

10. Do Exercise 1.20 from the text.

11. Do Exercise 1.21 from the text.