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 $2, 3, 4, 5, 6, \ldots$, but rather $2, 3, 5, 7, 9, \ldots$. 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?

Answer. Instructed by the description in the problem, we may immediately write down our defintion of `next` as follow:

```
(define (next n)
  (if (= n 2)
      3
      (+ n 2)))
```

With this, it becomes easy for us to rewrite a more advanced version of `smallest-divisor`:

```
(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 (next divisor)))))
```

Now, we have to run our new `smallest-divisor` and compare its performance with that of the former version. If everything goes smoothly, you might find out that the result negates our pridiction. The new `smallest-divisor` runs only about 1.69 times as fast on average, not twice, as is shown in Table 1.

Magnitude	Prime	Running Time(Former Version)	Running Time(Updated Version)	Ratio
10^9	1000000007	0.05000000000000071	0.02999999999999936	1.66666666666673
	1000000009	0.03999999999999915	1.9999999999999574e-2	2.0
	1000000021	4.0000000000000924e-2	1.9999999999999574e-2	2.00000000000009
10^{10}	10000000019	0.13000000000000078	0.08000000000000007	1.62500000000001
	10000000033	0.11999999999999922	0.08000000000000007	1.49999999999999
	10000000061	0.13000000000000078	0.08000000000000007	1.62500000000001
10^{11}	100000000003	0.3999999999999986	0.25	1.59999999999999
	100000000019	0.41000000000000014	0.2400000000000002	1.70833333333333
	100000000057	0.41000000000000014	0.25	1.64
10^{12}	1000000000039	1.2799999999999994	0.7699999999999978	1.66233766233767
	1000000000061	1.2999999999999972	0.7899999999999991	1.64556962025316
	1000000000063	1.2900000000000027	0.7899999999999991	1.63291139240507

Table 1. Comparison of Performance between Two Versions of `smallest-divisor`

Well, in order to blame what causes this strange phenomenon, let's try to focus on the fragment we changed just before. In a careful observation, one might find out that we updated our `smallest-divisor` by replacing the expression `(+ test-divisor 1)` with `(next divisor)` which involves a compound procedure `next`. Every time `(next divisor)` is evoked, it evalutes an `if` conditional, thus, additional evaluation has to be performed. Whereas, in `(+ test-divisor 1)` the operator is a primitive. Therefore, our new `smallest-divisor` does not run about twice as fast as we expected.