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.

```
(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 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 $\Theta(n)$, you should expect that testing for primes around 10,000 should take about 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?

Answer. [1]Given the procedure **timed-prime-test** in the problem, it takes us a little bit of inginuity to come up with the following **search-for-primes** procedure:

```
(define (search-for-primes lower upper)
  (cond ((> lower upper)
         (newline)
         (display "Finished in testing primality."))
        ((even? lower) (search-for-primes (+ lower 1) upper))
        ((timed-prime-test lower)
         (search-for-primes (+ lower 2) upper))))
```

Now, using our **search-for-primes** procedure defined aboved, let's try to find out the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000[2]:

```
(search-for-primes 1000 1019)
1009 *** 0.
1013 *** 0.
1019 *** 0.

(search-for-primes 10000 10037)
10007 *** 0.
10009 *** 0.
10037 *** 0.

(search-for-primes 100000 100043)
100003 *** 0.
100019 *** 0.
100043 *** 0.

(search-for-primes 1000000 1000037)
1000003 *** 0.
1000033 *** 0.
1000037 *** 0.
```


1. Solution for this exercise was implemented and tested in MIT/GNU Scheme (Release 9.1.1) on a MacBook Pro running OS X 10.8.3.

2. Note that here I've leave out the trivial output of composite numbers for clarity. To see the actural output given by the interpreter, please refer to the file "Test_for_Exercise_1.22.scm". So does the following output for the same reason.

Nowadays, computers have become too fast to appreciate the accuracy of testing such relatively small numbers given by the problems(the tests above really bear this out). So, in order to obtain more acceptable results, we are supposed to enlarge our testing data by a magnitude of, for example, 10^6.

```
(search-for-primes 1000000000 1000000021)
1000000007 *** .05000000000000071
1000000009 *** .03999999999999915
1000000021 *** 4.0000000000000924e-2

(search-for-primes 10000000000 10000000061)
10000000019 *** .13000000000000078
10000000033 *** .11999999999999922
10000000061 *** .13000000000000078

(search-for-primes 100000000000 100000000057)
100000000003 *** .3999999999999986
100000000019 *** .41000000000000014
100000000057 *** .41000000000000014

(search-for-primes 1000000000000 1000000000063)
1000000000039 *** 1.2799999999999994
1000000000061 *** 1.2999999999999972
1000000000063 *** 1.2900000000000027
```

Given the timing data by the interpreter, we can immediately form a table to check the pridiction of the order of growth, as is shown in Table 1.

Lower Bound	Prime	Time (s)	Average Time (s)	Ratio
10^9	1000000007	0.05000000000000071	0.0433333333333336	–
	1000000009	0.03999999999999915		
	1000000021	4.0000000000000924e-2		
10^{10}	10000000019	0.13000000000000078	0.126666666666667	2.92307692307691
	10000000033	0.11999999999999922		
	10000000061	0.13000000000000078		
10^{11}	100000000003	0.3999999999999986	0.406666666666666	3.21052631578946
	100000000019	0.41000000000000014		
	100000000057	0.41000000000000014		
10^{12}	1000000000039	1.2799999999999994	1.29	3.17213114754099
	1000000000061	1.2999999999999972		
	1000000000063	1.2900000000000027		

Table 1. Time Required to Test The Primality in Different Magnitude

Well, the last comlumn in Table 1 indicates that the ratio of the average time for finding the first 3 prime numbers between adjcent magnitude varies around $\sqrt{10}$, which is approximately 3.162. Therefore, our result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation.