Prime Testing (Basic Algorithm)

You may have noticed that you can use your previous program to identify prime numbers: if a number has exactly two divisors, then it must be prime. However, counting all of the divisors of a number to decide whether it is prime is very inefficient. Consider, for example, the number 123,457,791,132. Is it prime? Obviously not, since it is an even number and, therefore, divisible by 2. By recognizing that this number is not prime because it is divisible by 2, you save yourself over 123 billion divisions!

There is a second way to shorten the testing process. The thing to notice is that divisors (usually) come in pairs, one big, one small. Consider, for example, the divisors of 900:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>900</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>225</td>
</tr>
<tr>
<td>5</td>
<td>180</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>90</td>
</tr>
<tr>
<td>12</td>
<td>75</td>
</tr>
<tr>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>18</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>45</td>
</tr>
<tr>
<td>25</td>
<td>36</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

The only exception is the divisor 30 which is paired with itself, because 30 is the square root of 900.

Our conclusion is that whenever you find a divisor less than the square root of an integer \( n \), there is a corresponding divisor greater than the square root of \( n \). Thus, we need only look for divisors between 1 and \( \sqrt{n} \). If there are none, then there are also none between \( \sqrt{n} \) and \( n \). This saves considerable searching! For example, if \( n = 1234357 \), then we need only check for divisors between 1 and \( \sqrt{1234357} \), which is less than 1112. In other words, we only have to check a little over a thousand divisors, rather than over a billion.
Here’s a simple program to test whether a number is prime by checking all possible divisors.

```java
/*
  Date   Name  PrimeTest  Simple Primality Test using Trial Division
*/
public class PrimeTest {
  public static boolean primetest(long n) {
    if (n <= 1) return(false);
    boolean AmIPrime = true;
    for (int i = 2; i < n; i++) {
      if (n % i == 0) {
        AmIPrime = false;
        break;
      }
    }
    return(AmIPrime);
  }
  public static void main(String[] args) {
    long n = Long.parseLong(args[0]);
    if (primetest(n))
      System.out.println(n + " is prime. ");
    else
      System.out.println(n + " is not prime. ");
  }
}
```

Try the following experiments.

1. Enter and run the program. Does it work? Try some numbers you know are prime and some you know are not.

2. Modify the program to use the shorter method (testing fewer possible divisors) discussed above, and try it on a large number found to be prime with your previous version. Does it work? Is it noticeably faster? [Hint: you do not need to compute a square root!]

---

Copyright © 2019, 2020 by Walter Carlip and the University of Chicago Young Scholars Program
3. Change the `main` method to loop through the integers from 1 to 1000 and print out only those that are prime. Is the program faster than your earlier program?

4. The number 1 is not prime, 11 is prime, 111 is not prime. Are any other of the “repunit” numbers, that is, numbers that are written as a sequence of repeated ones, prime? Change your program to loop and check each repunit to see if it is prime. [Hint: To get from one to the next, multiply by ten and add one.]