Lab 6a PrimesProper
Loops, Arrays, Methods

This lab revisits what you did with prime numbers in Lab 5. If you remember, you were asked to give the nth prime number given a user input. You probably wrote a loop to do this. This lab takes the same idea but is meant to be more efficient and uses some properties of prime numbers which are amazingly cool if you ever get to take a number theory class.

It used to be believed (back in the 1960s and 1970s) that primes were cute and fun to play with and not much else, but then with the advent of computers on a large scale and transfers of money and information electronically that had to be protected, all of the sudden prime numbers became very important.

It turns out that it is very difficult to find the prime factors (those primes that are multiplied together) of a very large number, and this is what has to be done to crack encrypted information. And by large I mean numbers with 200+ digits. Seems like a computer should be able to do this pretty fast, but it turns out that the fastest computers in the world balk at trying to solve this. For example, the RSA encryption algorithm uses 128 bit keys(the key is what is used to encrypt your information). This requires 2^{128} operations (like divide, multiply) to decode those numbers. Now, suppose you have a computer that can do 100 billion operations/sec (that would be a 100 gigahertz computer). That number is $100x10^9$. Now, 2^{128} is roughly $(10^9)^4$. That is a billion to the 4^{th} power, or 10^{36}.

So, doing a little math: $10^{36}/(100x10^9)$ is 10^{26}. That means 10^{26} seconds best case to decompose a number to its prime factors. There are 3600 x 24 x 365 seconds in a year. That number is roughly 31 million seconds per year. So, $10^{26}/(3.1x10^7)$ is roughly a whooping $3x10^{18}$ years or longer than the universe has existed. This is why it is safe (today) to use https sites. They encrypt everything using really big numbers.

But, this is a complete digression.

What I want you to do:

Write a program that uses an array to store the first one thousand primes. This array should be able to hold **long** integers. You will write a method called from **main** and the method must be called:

fillprimearray(long [] primearray)

Your program will start up and the **main** method will call the above method passing the **primearray that is to be declared local to main**. The routine then fills up the array (see below). Once the array is filled, the main method will prompt the user to ask for a particular prime (like maybe the 31st). The main method will then look up in cell (30) of the array for that prime and print it out. This loop should run until the user asks for the -1th prime.

Now, filling the array. You need to start counting numbers from 2 up to whatever and putting them in the array if they are prime. To see if the next number is prime you only have to see if any of the primes you already have divide it evenly. If they do, then the new number isn't prime. This is a beautiful property of prime numbers! Every other number must be constructed from multiplying prime numbers together! Wow!

So, this will force you to use arrays and loops and you will see the enormous value of using an array.

As an example:

Let's pretend I have a small array holding the first five primes:
2, 3, 5, 7, 11

and I'm trying to fill in the next prime. Since primes have to be odd (except for the value **2)**, then the next number I look at is 13. I try to divide 13 by all the primes in the array up to 13/2 (why do I only have to go halfway????). So, I try to divide by 2 then 3 then 5 and then I'm done. None of them divide it so it must be prime!

Now my array holds
2, 3, 5, 7, 11, 13

and the next number to try is 15.

I divide 15 by 2 and get 7. So, I try to divide 15 by all the numbers up to 7 and discover that 5 divides it evenly. So 15 doesn't go in the array and I start work with 17.

Get the idea? Get to work!