This programming assignment is about working with sets in Java. It should be finished and turned in by Friday, October 8, the last day of classes before Fall break. Programming assignments differ from homework assignments in that ***you should work on the programs on your own.*** *Furthermore, you should not use code from the Internet or other sources. You should develop and write the program yourself. You can, of course, ask me for help.* ***Submit your programs to your homework folder in the directory*** `/classes/f10/cs229/homework`.

The **Sieve of Eratosthenes** is a classic algorithm for finding all the prime numbers less than or equal to some given integer N. It can be stated very simply in pseudocode: To construct the set of prime numbers less than or equal to `MAX_N`:

```
Let primes be the empty set.
for (int n = 2; n <= MAX_N; n++)
   add n to primes
for (int p = 2; p <= MAX_N/2; p++) {
   if (p is in primes) {
      for (int n = 2*p; n <= MAX_N; n += p)
          remove n from primes [if it's in primes]
   }
}
```

When this algorithm has finished, the integers that remain in the set *primes* are all the prime numbers less than or equal to `MAX_N`.

For this assignment, you will write four implementations of the Sieve, using different representations for sets of integers. The goal is really to learn about sets in Java, not to find prime numbers.

The set representations that you should use are: (1) Use the type *TreeSet<Integer>*. (2) Use an array of *boolean*. (3) Use the type *BitSet*. (4) Use an array of *int*, using just one bit for each potential member of the set. In each program, you should use a constant or an input number to represent the upper limit `MAX_N`. The program should run the Sieve algorithm, timing how long it takes, and output the number of seconds. It should then output the number of primes less than or equal to `MAX_N`; this is just the number of integers in the set at the end of the algorithm. As a check on correctness, it should also output a list of the first few elements of the set (say, the elements that are less than 100).

For representation (4), it is possible to make `MAX_N` larger than 2^{31}, provided that it is declared to be of type *long*. This will require some type-casts to *int* later in the program, since the length of an array and an index for an array must be of type *int*. For some extra credit, you can write a program that can be run with `MAX_N` equal to ten billion. You should turn in a statement of the number of primes less than ten billion and how long it took for your program to find them. (My program took 616 seconds on my home computer.)

The Java class *Sieve.class* in the directory `/classes/f10/cs229` is one implementation. You can run it to see how the program should behave. In this version, `MAX_N` is of type *int*, and so cannot be larger than about two billion.

Here is some additional information on each version of the program:

1. To use the data type *TreeSet<Integer>*, your program should `import java.util.TreeSet`. If *primes* is a variable of type *TreeSet<Integer>*, you can create an initially empty set with a call to the constructor

```
primes = new TreeSet<Integer>();
```

To add an integer x to the set, call `primes.add(x)`. To remove x from the set (if present), use `primes.remove(x)`. To test if x is in the set, call the boolean-valued function `primes.contains(x)`. The number of elements of the set is returned by the function `primes.size()`.

TreeSet<Integer> is a natural representation for sets of integers. It has methods that directly implement the common set operations that you need for this program. However it uses a lot of memory and is less efficient than more direct methods. Don't expect to use use extremely large values of `MAX_N` with this representation.

2. A *boolean* array is a less direct, but still fairly easy, way to represent a set. For a set of integers in the range 0 to `MAX_N`, use an array, *primes*, of `MAX_N+1` booleans. In this representation, `primes[x]` is true if and only if x is in the set. The implementation of the Sieve algorithm is straightforward, except that there is no built-in method for finding the number of elements in the set.

Although one bit should suffice to represent a *boolean* value, a *boolean* array actually uses one byte of memory for each element of the array. Still, with gigabytes of memory to play with, the real limitation is that an array index must be an *int*. This limits `MAX_N` to no more than 2^{31}, or about two billion.

3. The *BitSet* class represents sets of integers in a way that uses only one bit per possible element, so it is more memory efficient than using boolean arrays. To use it in your program, you should `import java.util.BitSet`. If *primes* is a variable of type *BitSet*, you can create an object to represent sets of natural numbers less than `MAX_N` with

```
primes = new BitSet(MAX_N);
```

To add an integer x to the set, call `primes.set(x)`. (There is a more efficient way to add a lot of elements at once; see the API.) Use `primes.clear(x)` to remove x from the set. To test whether x is the set, use the boolean-valued function `primes.get(x)`, which returns *true* if x is in the set. The number of elements of the set can be determined by calling `primes.cardinality()`.

With all these methods, it's easy to implement the Sieve of Eratosthenes with a *BitSet*. However, the elements of the set are limited by the *BitSet* API to be of type *int*, so you are still limited to numbers no larger than 2^{31}.

4. For the final representation, you have to implement sets by hand, in essentially the same way that it is done in a *BitSet*, that is, using just one bit for each possible element of the set. For this representation, you can use an array of *int*. You will need to use the bitwise operators that are discussed in Section 2.3 of the textbook.

Each *int* in the array has 32 bits and can be used to represent 32 potential elements of the set. Given a potential element x, the array index for x can be computed as `x / 32` or, better, as `x >> 5`. This gives you an *int*, but you still need the correct bit position for x within that *int*. That can be computed as `x % 32` or, better, as `x & 31`.

To perform the operations of adding and removing elements and to test whether an element is in the set, you will have to use Java's bitwise operations. This is certainly the most challenging part of the assignment.