THE SIEVE OF ERATOSTHENES

An ancient Greek mathematician by the name of Eratosthenes of Cyrene (c. 200 B.C.) developed an algorithm for finding prime numbers that has come to be known as the *Sieve of Eratosthenes*.

<table>
<thead>
<tr>
<th>Eratosthenes’s Sieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a sieve containing all the integers from 2 through $n$.</td>
</tr>
<tr>
<td>2. Remove the nonprime integers from the sieve by removing the multiples of 2, of 3, of 5, and so on, until reaching $\sqrt{n}$.</td>
</tr>
<tr>
<td>3. Only primes will remain in the sieve.</td>
</tr>
</tbody>
</table>

Let’s design a Java program that uses this algorithm. We’ll follow the programming methodology *How to Invent an Algorithm*.¹

<table>
<thead>
<tr>
<th>How to Invent an Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Let $n = 35$. Create a sieve containing 2 through 35:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

Remove the multiples of 2:

2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35

The multiples of 3:

2 3 5 7 11 13 17 19 23 25 29 31 35

The multiples of 5:

2 3 5 7 11 13 17 19 23 29 31 35

$\sqrt{35} > 5.92 = \sqrt{35}$, so we’re done. The numbers left are all prime.

How to Invent an Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>What?</th>
<th>How?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Determine the program’s requirements</td>
<td>• Identify the program’s input and output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Determine what computations transform the input into the output.</td>
</tr>
</tbody>
</table>

INPUT

```
\[ n \rightarrow \text{Sieve of Eratosthenes} \rightarrow \text{OUTPUT} \]
```

OUTPUT

```
\[ \text{prime}_1, \text{prime}_2, \ldots, \text{smallest prime} < n \]
```

How to Invent an Algorithm – Help for Beginners

<table>
<thead>
<tr>
<th>Step</th>
<th>What?</th>
<th>How?</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Use top-down design to develop a pseudo-code algorithm that solves the problem</td>
<td>• Write down a first-draft algorithm as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>o Write in English the steps you employed when solving the problem by hand.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>o Don't worry about language or details; just get your ideas on paper in outline form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>o Be sure to include the input and output steps.</td>
</tr>
</tbody>
</table>

Algorithm for Sieve of Eratosthenes (1st draft)

```
input n
create an array containing the numbers 2 through n
remove the multiples of 2 from the array
remove the multiples of 3 from the array
remove the multiples of 5 from the array
remove the multiples of 7 from the array
continue until reaching the \[ \sqrt{n} \]
print the contents of the array
```
How to Invent an Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>What?</th>
<th>How?</th>
</tr>
</thead>
</table>
| 3    | Use top-down design to develop a pseudo-code algorithm that solves the problem | • Introduce program variables to hold data you’ve identified.  
• Identify and clarify any loops that you need.  
• Use pseudo-code. |

Algorithm for Sieve of Eratosthenes (1st draft)

```
input n
create an array containing the numbers 2 through n
!
if k is in the array
    remove the multiples of k from the array
k++
remove the multiples of k from the array
remove the multiples of 5 from the array
continue until reaching \( \sqrt{n} \)
!
print the contents of the array
```

Algorithm for Sieve of Eratosthenes (2nd draft)

```
input n
create an array containing the numbers 2 through n
!
for k = 2, 3, ..., \( \sqrt{n} \) do
    if k is in the array then
        remove the multiples of k from the array
    end if
end for
!
print the contents of the array
```
Suppose \( k \) is 2. Let’s remove the multiples of 2 from the array.

The first multiple of 2 is \( 2+2 = 4 \). Remove it:

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

The second multiple is \( 4+2 = 6 \). Remove it:

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

Continue with \( 6+2 = 8, 8+2 = 10 \), etc.

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

Stopping when reaching the end of the array at 35:

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

The process is the same for \( k = 3, 5 \), etc.

**Algorithm for remove the multiples of \( k \) from the array**

- suppose \( k = 2 \)
- calculate the first multiple of 2 as \( 2+2 = 4 \)
- remove 4 from the array
- calculate the next multiple of 2 as \( 4+2 = 6 \)
- remove 6 from the array
- calculate the next multiple of 2 as \( 6+2 = 8 \)
- continue until reaching \( n \)
Algorithm for remove the multiples of $k$ from the array

- calculate the first multiple of $k$ as $2 	imes k = 2k$
- remove $2k$ from the array
- calculate the next multiple of $k$ as $(2k) + k = 3k$
- remove $3k$ from the array
- calculate the next multiple of $k$ as $(3k) + k = 4k$
- continue until reaching $n$

LOOP
- quit when $m$ reaches $n$

Algorithm for Sieve of Eratosthenes (3rd draft)

- input $n$
- create an array containing the numbers 2 through $n$
- for $k = 2, 3, \ldots, \sqrt{n}$ do
  - if $k$ is in the array then
    - $m = k + k$
    - while $m \leq n$ do
      - remove $m$ from the array
      - $m = m + k$
    - end while
  - end if
- end for
- print the contents of the array
To code this algorithm, we need to give some thought to the implementation of the sieve. The algorithm reveals the properties of the sieve when viewed as a data abstraction.

### The sieve Data Abstraction

<table>
<thead>
<tr>
<th>What is it?</th>
<th>A container of numbers holding 2 through n</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is done to it?</td>
<td>create it  check if k is in it  remove m from it</td>
</tr>
</tbody>
</table>

Novice programmers immediately choose to implement the sieve by storing the numbers in an array, as with:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
```

Seasoned programmers, however, are more careful. They know that the implementation of a data abstraction can have a great effect on the complexity and performance of the final program. Here, for example, are some complexities of the novice solution.

### Complexities of Implementing the sieve Data Abstraction as an Array of Numbers

| check if k is in it | • Do we have to look through the entire array to find it? |
| remove m from it    | • How, exactly do we “remove” a number?     |
|                     | • What do we do with the empty array space that held it? |

Looking for a better implementation, notice that multiples are spaced evenly apart. For example, every other number is even:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
```

Every third number is divisible by 3:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
```

Every fifth number is divisible by 5:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
```
This suggests that we can use the indices of the array to represent the numbers rather than the contents of the array. Make the elements of the array true or false. Position \( k = \text{true} \) means that \( k \) is in the sieve; position \( k = \text{false} \) means it isn’t. Ignore the first two elements (positions 0 and 1) since we never have those numbers in the sieve.

This implementation greatly simplifies the implementation of the algorithm.

<table>
<thead>
<tr>
<th>Implementing the sieve Data Abstraction as a Boolean Array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>create it</strong></td>
</tr>
<tr>
<td><strong>check if ( k ) is in it</strong></td>
</tr>
<tr>
<td><strong>remove ( m ) from it</strong></td>
</tr>
</tbody>
</table>

To initially place numbers into the sieve, initialize the corresponding array values to true. The first two elements of the array aren’t used.

Remove the multiples of 2 by setting every other element to false:

Remove the multiples of 3 by setting every third element to false:

Remove every fifth element:
import javax.swing.JOptionPane;

public class Sieve
{
    public static void main( String [] args )
    {
        String prompt = "Enter positive integer: ";
        String input = JOptionPane.showInputDialog( prompt );
        // halt if user cancels input dialog
        if ( input == null ) System.exit( 0 );
        // convert input string to an int
        int n = Integer.parseInt( input );
        // create and initialize the sieve
        boolean [] sieve = new boolean[n+1];
        sieve[0] = sieve[1] = false;
        for ( int k = 2; k <= n; k++ )
        {
            sieve[k] = true;
            // remove the nonprimes
            for ( int k = 2; k*k <= n; k++ ) // quit when k is sqrt(n)
            {
                if ( sieve[k] ) // k is in sieve
                {
                    // begin with 2k, the first multiple
                    int m = k + k;
                    while ( m <= n )
                    {
                        sieve[m] = false;
                        m += k; // add k to get next multiple
                    }
                }
            }
        }
        // print whatever is left in the sieve
        System.out.println( "A list of prime numbers from 2 to " + n );
        int count = 0;
        for ( int k = 2; k <= n; k++ )
        {
            if ( sieve[k] )
            {
                System.out.println( ++count + ": " + k );
            }
        }
    }
}
Exercises

Enter the Sieve application on the previous page into jGRASP and save it to a file. Compile it and fix any syntax errors. Perform the following series of experiments and answer any questions.

1. In jGRASP’s main window, set a breakpoint at the line shown below. To set the breakpoint, move the mouse cursor to the gray bar at the left of your code. It will display a red dot •. Click the mouse to set the breakpoint. Set a breakpoint at line 14:

   ```java
   boolean [] sieve = new boolean[n+1];
   ```

2. Start jGRASP’s debugger by clicking the Ladybug button 🐞. Execution will pause at line 8:

   ```java
   String input = JOptionPane.showInputDialog( prompt );
   ```

   Enter the input 35.

3. After entering the input, program execution halts at line 14:

   ```java
   boolean [] sieve = new boolean[n+1];
   ```

   Notice that program execution halts immediately before the statement has been executed.

   In jGRASP’s Debug pane, what is the value of variable n?

4. Open a jGRASP Viewer object to view the sieve array. To do this: move the mouse to the Debug pane, point it at sieve’s blue square 🔄 sieve, click, hold and drag it away from the Debug pane.

   Once the jGRASP Viewer is open, resize it as needed and adjust the Scale and Width slider controls to show the entire array. The Viewer should appear similar to this:
5. In jGRASP’s main window, set another breakpoint at the line shown below:

   \[\text{for} \ ( \text{int} \ k = 2; \ \text{k}\times\text{k} \leq n; \ k++ ) \ \text{// quit when k is sqrt(n)}\]

Resume execution by clicking jGRASP’s Resume button ➤.

In jGRASP’s Viewer, what has the program done to the sieve array?

6. Use jGRASP’s Step button ⬇️ to step through several cycles of the program’s while loop.

In jGRASP’s Debug pane, what is the program doing to variables \(k\) and \(m\)?

In jGRASP’s Viewer, what is the program doing to the sieve array?

7. Resume execution by clicking jGRASP’s Resume button ➤. The program completes the first cycle of the for loop and halts again at this line:

   \[\text{for} \ ( \text{int} \ k = 2; \ \text{k}\times\text{k} \leq n; \ k++ ) \ \text{// quit when k is sqrt(n)}\]

In jGRASP’s Debug pane, what is the value of variable \(k\)?

In jGRASP’s Viewer, what has the program done to the sieve array?

8. In jGRASP’s main window, set another breakpoint at the line shown below:

   \[\text{int} \ \text{count} = 0;\]

Repeat the instructions in exercises 6 and 7, following the program’s computation until reaching the line above.

6. Use jGRASP’s Step button ⬇️ to step through several cycles of the program’s for loop.

In jGRASP’s main window, observe the progress of the computation and in the Run I/O pane observe the output being generated.

7. Finish the program by clicking jGRASP’s Resume button ➤.