**Algorithm Study Template**

**Algorithm**: The Sieve of Eratosthenes

**aka**: <no alternative names>

**Techniques**: Sieving

**Categories**: Searching, Prime Number Sieve

**Problem**: The Sieve of Eratosthenes is used to find all prime numbers up to a given limit. For example, the sieve can be used to determine that there are four prime numbers less than or equal to the number ten.

**Applications**: Via Wikipedia, “(Algorithms that generate prime numbers) are used in various applications, (such as) hashing, public-key cryptography, and (searching for) prime factors in large numbers.”

A more specific example would be finding prime factors of the number one million. To do this quickly, you could use the Sieve of Eratosthenes to generate all primes less than or equal to one million and divide one million by each of those primes. The primes which divide evenly can be stored somewhere as prime factors.

**References**:

* <http://mathworld.wolfram.com/PrimeNumber.html>
* <http://xlinux.nist.gov/dads/HTML/sieve.html>
* <http://motivate.maths.org/content/generating-prime-numbers-0>
* <http://www.cut-the-knot.org/Curriculum/Arithmetic/Eratosthenes.shtml>
* <http://primes.utm.edu/glossary/xpage/sieveoferatosthenes.html>
* <http://mathworld.wolfram.com/SieveofEratosthenes.html>
* <http://rosettacode.org/wiki/Sieve_of_Eratosthenes>
* <http://www.algolist.net/Algorithms/Number_theoretic/Sieve_of_Eratosthenes>
* <http://demonstrations.wolfram.com/SieveOfEratosthenes/>

**Implementation details**:

* **Big Idea**: To find all prime numbers less than or equal to a user-specified limit called n, we create a list of integers from two to n where all entries are initially marked prime and then iterate through multiples of primes, marking them as non-prime. The remaining integers marked prime are the only primes from two to n.
* **Description**: Via Wikipedia:

To find all the prime numbers less than or equal to a given integer *n* by Eratosthenes' method:

1. Create a list of consecutive integers from 2 to *n*: (2, 3, 4, ..., *n*).
2. Initially, let *p* equal 2, the first prime number.
3. Starting from *p*, count up in increments of *p* and mark each of these numbers greater than *p* itself in the list. These will be multiples of *p*: 2*p*, 3*p*, 4*p*, etc.; note that some of them may have already been marked.
4. Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

The main idea here is that every value for *p* is prime, because we have already marked all the multiples of the numbers less than *p*.

* **Pseudo-code**: Via Wikipedia:

**Input**: an integer *n* > 1

Let *A* be an array of Boolean values, indexed by integers 2 to *n*, initially all set to **true**.

**for** *i* = 2, 3, 4, ..., not exceeding *√n*:

**if** *A*[*i*] is **true**:

**for** *j* = *i2*, *i2+i*, *i2+2i*, ..., not exceeding *n* :

*A*[*j*] := **false**

Now all *i* such that *A*[*i*] is **true** are prime.

* **Specific implementation**: (see Eratosthenes.java)

**Correctness**:

**Theoretical**: Beginning from two, it is obvious that if we mark off multiples of two, the lowest unmarked number must be a prime, thus three is the next prime. Repeating the process with each prime discovered will lead to the next prime until all primes up to the given limit, n, have been found. At this point, all non-primes have been marked as such and only the primes remain unmarked.

**Empirical**:

For testing purposes, the program allows the user to input an integer greater than one. The program then uses a try-catch statement to find input mismatch exceptions such as strings or doubles. If such an exception is found, the user is notified that their input was incorrect, and the program terminates.

After receiving acceptable input, the program then finds the number of primes less than or equal to the user-specified limit by using the Sieve of Eratosthenes and a separate brute-force method. These methods are run separately, but yield identical results. The brute-force search is run specifically for performance testing, as seen below. In order to ensure correctness, the numbers marked as prime by both methods are subjected to a primality test. If that test were to fail for any number marked as prime by either method, an error message would be generated and the program would terminate. In addition, after the primality test is completely passed, the program checks to make sure that each method yielded the same number of primes less than or equal to the limit. If that test were to fail, once again, an error message would be generated and the program would terminate.

One may wonder why the user is required to enter an integer greater than one. This is because there are no prime numbers less than two, so checking for primes less than or equal to one or any integer less than one is pointless. Thus, two is the lowest integer the user is allowed to enter.

Specific test results:

Prime numbers <= 1,000,000,000 (1 billion):

The Sieve of Eratosthenes determined that there are 50,847,534 prime numbers <= 1,000,000,000.

The brute-force search also determined that there are 50,847,534 prime numbers <= 1,000,000,000.

Prime numbers <= 100,000,000 (one hundred million):

The Sieve of Eratosthenes determined that there are 5,761,455 prime numbers <= 100,000,000.

The brute-force search also determined that there are 5,761,455 prime numbers <= 100,000,000.

Prime numbers <= 10,000,000 (ten million):

The Sieve of Eratosthenes determined that there are 664,579 prime numbers <= 10,000,000.

The brute-force search also determined that there are 664,579 prime numbers <= 10,000,000.

Prime numbers <= 1,000,000 (one million):

The Sieve of Eratosthenes determined that there are 78,498 prime numbers <= 1,000,000.

The brute-force search also determined that there are 78,498 prime numbers <= 1,000,000.

**Performance**:

**Theoretical**: The complexity of the Sieve of Eratosthenes is *O*(*n*(log*n*)(loglog*n*)) bit operations with a memory requirement of *O*(*n*).

**Empirical**: The following are time measurements taken during the execution of the Sieve of Eratosthenes and the brute-force method. Note that the times do not include execution of primality tests, printing output, or the same-number result test. The times are specifically meant to show the duration of execution for the algorithmic steps of finding primes and counting them (as seen in the above pseudo-code for the Sieve of Eratosthenes).

Execution times:

Prime numbers <= 1,000,000,000 (1 billion):

The Sieve of Eratosthenes was completed in 18,139.217 milliseconds.

The brute-force search was completed in 5,258,160.147 milliseconds.

Prime numbers <= 100,000,000 (one hundred million):

The Sieve of Eratosthenes was completed in 2,594.399 milliseconds.

The brute-force search was completed in 188,800.607 milliseconds.

Prime numbers <= 10,000,000 (ten million):

The Sieve of Eratosthenes was completed in 112.635 milliseconds.

The brute-force search was completed in 7,385.519 milliseconds.

Prime numbers <= 1,000,000 (one million):

The Sieve of Eratosthenes was completed in 25.369 milliseconds.

The brute-force search was completed in 314.041 milliseconds.

One can easily see that the Sieve of Eratosthenes is much faster than a brute-force method for finding large numbers of primes. However, the performance gap becomes smaller as the number of primes does. This is likely because the number of divisibility tests grows very quickly for a brute-force method as the limit, n, becomes larger. The brute-force method I used is optimized to only test for primes not exceeding the square root of the limit, n, but still takes a long time for large values of n. It is also noteworthy that the Sieve of Eratosthenes finds composite numbers by marking multiples of primes instead of testing for divisibility, as a brute-force method does. That difference in strategy helps illustrate the performance edge of the Sieve of Eratosthenes and makes a good case for its use.

(To reiterate, my inclusion of a brute-force method is solely to show that the Sieve of Eratosthenes performs better by an order of magnitude in cases where large numbers of primes are being generated.)

**Anecdotes**: Anticipating that the test for primes less than or equal to one billion would take a long time, I started it and went to bed. When I checked on it in the morning, I was happy to see that the test had successfully completed. Because of the primality tests, I’m still not sure exactly how long it took for the program to finish execution.

**History**: Via Wikipedia: “ (The Sieve of Eratosthenes) is named after Eratosthenes of Cyrene, a Greek mathematician; although none of his works have survived, the sieve was described and attributed to Eratosthenes in the *Introduction to Arithmetic* by Nicomachus.”

**Variations**: Euler’s Sieve is a variation of the Sieve of Eratosthenes in which each composite number is marked as non-prime only once by removing composites from the list during execution of the sieve.

**Alternatives**: The Sieve of Atkin and Sieve of Sundaram can also be used to find all prime numbers up to a specified integer. The Sieve of Atkin is an optimized version of the Sieve of Eratosthenes. The Sieve of Sundaram is similar to the Sieve of Eratosthenes, but handles even numbers in a unique way.

**Credits:**

* <http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes>
* <http://en.wikipedia.org/wiki/Generating_primes>
* <http://answers.yahoo.com/question/index?qid=20080923073045AABvpVz>
* <http://www.dreamincode.net/forums/topic/165561-checking-to-see-if-a-number-is-prime/>
* <http://stackoverflow.com/questions/7991463/what-is-the-best-most-performant-algorithm-to-find-all-primes-up-to-a-given-num>
* I would like to credit John Lubke, a fellow classmate, for bringing the Sieve of Eratosthenes to my attention by posting about it on the class discussion group and piquing my interest in it.