The Sieve of Eratosthenes

Robert J. Hansen

April 8, 2017

Contents

1 The Sieve of Eratosthenes

The mathematicians of antiquity discovered many ways of determining prime numbers. The most famous is the Sieve of Eratosthenes, which may be misnamed. Its first known mention comes from Nichomachus' *Introduction to Arithmetic,* wherein it is attributed (without supporting evidence) to Eratosthenes of Cyrene.

2 The algorithm

In its broadest strokes, it involves writing the numbers from 1 to n in a square. The number 1 is immediately blocked out. Moving on to the number 2, every following multiple of 2 is blocked out. Moving on to the number 3, every following multiple of 3 is blocked out. Moving on to the number 4, we discover it was blocked out by 2, and so we move on to the number 5, blocking out its following multiples. Once we reach the end of the first row of the square, all the primes within the square have been discovered.

From this we may discern the following algorithm:

1. Start with a vector V of numbers 2 through N.
2. Determine a stopping-point S of $\sqrt{N}$.
3. Point an iterator *iter* to the first element of N.
4. If *iter* is invalid, or if the pointed-to element is greater than S, terminate.
5. Otherwise, remove from the vector all integer multiples of the pointed-to element.
6. Move to the next element in the vector.

3 The algorithm in C++

C++ allows us to implement the Sieve of Eratosthenes in a meaningful and straightforward way.

3 ⟨ Headers and usings 4 ⟩
⟨ The Sieve of Eratosthenes 5 ⟩

3.1 Headers and usings

We make heavy use of the C++ standard library to express the algorithm in a terse and efficient style.

4 ⟨ Headers and usings 4 ⟩ ≡

```
#include <iostream>
#include <sstream>
#include <iterator>
#include <functional>
#include <algorithm>
#include <vector>
#include <cmath>
  using std::stringstream;
  using std::vector;
  using std::remove_if;
  using std::bind2nd;
  using std::not2;
  using std::modulus;
  using std::copy;
  using std::ostream_iterator;
  using std::cout;
  using std::cerr;
```

This code is used in chunk 3.

3.2 The Sieve of Eratosthenes

5 ⟨ The Sieve of Eratosthenes 5 ⟩ ≡

```
vector < unsigned int > psieve(const unsigned int N) {⟨ Constant
        initializations 6 ⟩
    ⟨ Vector initialization 7 ⟩
    ⟨ Sieve the vector 8 ⟩
    ⟨ Return a value 9 ⟩
    }
```

This code is used in chunk 3.

3.2.1 Constant initializations

Now armed with knowledge of the range over which we wish to find primes, we determine the range over which we must search. Once we've found primes up to the square root of our range, we can terminate. Since this value is invariant throughout the program's execution, we declare it **const**.

6 ⟨ Constant initializations 6 ⟩ ≡

```
const unsigned int S = static_cast⟨unsigned
    int⟩(sqrt(static_cast⟨double⟩(N)));
```

This code is used in chunk 5.

3.2.2 Vector initialization

Since N and S are not known until run-time, we have no choice but to initialize the vector programmatically at run-time.

7 ⟨ Vector initialization 7 ⟩ ≡

```
vector < unsigned int > v; vector < unsigned int > ::iterator iter;
for (unsigned int i = 2; i < N; i += 1) v.push_back(i);
iter = v.begin( );
```

This code is used in chunk 5.

3.2.3 Sieving the vector

Due to the expressiveness of C++, this can be expressed in a single line of code. Note well the use of standard C++ algorithms and functions. Particularly:

- *remove_if* marks elements within a given range for removal if and only if they satisfy a predicate. It takes three arguments: the beginning of the range, the end of the range, and the predicate to be used. The interval is half-open: the first element in the range is searched, but not the last. Since a vector's *end*( ) method returns one past the end of the vector, our call to *remove_if* starts the vector element *after* what's currently being examined, and removes those elements that are integer multiples of $*iter$.

- The predicate is unfortunately the kind of thing that could be used as a weapon in some rough neighborhoods. It is important to note that it revolves around the *modulus* $<$ **int** $>$ templated function, which accepts two parameters: a number, and over which value the modulus of the number ought be computed. In this case, we use *bind2nd* to force the second argument of *modulus* $<$ **int** $>$ to be $*iter$, we post-increment *iter* to advance us through the loop, and *not2* is used to invert the results.

  Without the *not2*, a computation like 4 mod 2 would yield zero. That zero would be interpreted as *false*, meaning 4 would not be removed. With the *not2*, the logical meaning of the clause is inverted: it will evaluate as *true*, meaning 4 will be removed.

- Finally, note the call to $v.erase(\,)$. Above it was mentioned that *remove_if* only marks something for removal — it doesn't actually remove anything. Instead, the output of remove can be given to a vector's *erase*( ) method, which will reap the elements.

8 ⟨ Sieve the vector 8 ⟩ ≡

```
while ((iter ≠ v.end( )) ∧ (∗iter ≤ S)) v.erase ( remove_if (iter + 1, v.end( ),
    bind2nd ( not2 ( modulus < int > ( ) ) , ∗iter ++ ) ) , v.end( ) ) ;
```

This code is used in chunk 5.

3.2.4 Returning the data

9 ⟨ Return a value 9 ⟩ ≡

```
return v;
```

This code is used in chunk 5.

Index

List of Refinements

⟨ Constant initializations 6 ⟩ Used in chunk 5.
⟨ Headers and usings 4 ⟩ Used in chunk 3.
⟨ Return a value 9 ⟩ Used in chunk 5.
⟨ Sieve the vector 8 ⟩ Used in chunk 5.
⟨ The Sieve of Eratosthenes 5 ⟩ Used in chunk 3.
⟨ Vector initialization 7 ⟩ Used in chunk 5.