Kobayashi Maru Primes

gerryrzeppa Uncategorized May 5, 2018 4 Minutes

It has been said that *"Great engineering strikes a balance between competing objectives."* I believe this to be true, in both mechanical and software engineering. In the latter, the competing objectives are typically clarity, size, and speed.

So I thought it a great opportunity to apply this principle when a computer scientist in South America challenged me to write a Plain English program, based on the Sieve of Eratosthenes, that could count the number of primes less than 250,000 in less than a second (on a bottom-of-the-line computer from Walmart).

Now Plain English is not the fastest language on earth, because clarity trumps speed when the primary goal of a language is to be *natural*. But Plain English is a *reasonably* fast language, and by balancing competing objectives, we can usually find an acceptable solution to any problem laid before us.

In this case, we first needed to make the famous Sieve of Eratosthenes clear; easy to understand. So we started with Wikipedia's description of the algorithm…

> *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 through n: (2, 3, 4, …, n).*
> *2. Initially, let p equal 2, the smallest prime number.*
> *3. Enumerate the multiples of p by counting to n from 2p in increments of p, and mark them in the list (these will be 2p, 3p, 4p, …; the p itself should not be marked).*
> *4. Find the first number greater than p in the list that is not marked. If there is no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.*

…which in Plain English reads like this:

```
To make a list of prime numbers less than or equal to a number:
Create the list of consecutive integers from 2 to the number. \ wiki's #1
Get an entry from the list. \ wiki's #2
Loop. Mark higher multiples of the entry. \ wiki's #3
Get the entry for the next lowest possible prime. If the entry is not nil, repeat. \ wiki's #4
```

That takes care of the *clarity* objective. If you're wondering, the type definitions and support routines that make that actually work are equally clear:

```
An entry is a thing with a number and a non-prime flag.
A list is some entries.
```

```
To create a list of consecutive integers from a number to another number:
Privatize the number.
Loop.
If the number is greater than the other number, exit.
Create an entry for the number.
Append the entry to the list.
Add 1 to the number.
Repeat.

To create an entry for a number:
Allocate memory for the entry.
Put the number into the entry's number.
Clear the entry's non-prime flag.

To mark higher multiples of an entry:
Privatize the entry.
Loop.
Put the entry's next into the entry.
If the entry is nil, exit.
If the entry's number is evenly divisible by the original entry's number,
set the entry's non-prime flag.
Repeat.

To get an entry for the next lowest possible prime:
Put the entry's next into the entry.
If the entry is nil, exit.
If the entry's non-prime flag is set, repeat.
```

So clarity, good. Size, good (the executable is only 150k). But what about speed? A little over 4 minutes to make the whole list for numbers between 1 and 250,000. Not good. Hmm… What would Captain Kirk do? What if we sacrificed a little size for speed? Why, after all, do we keep calculating primes over and over again when they never change? So I ran the program again, this time with an additional routine like this…

```
To make a prime string from a list:
Put "N" into the prime string.
Loop.
Get an entry from the list.
If the entry is nil, exit.
If the entry's non-prime flag is set, append "N" to the prime string; repeat.
Append "Y" to the prime string.
Repeat.
```

…to make a string with all the primes marked. Then I saved the string in a text file and pasted it into a library I called "Super fast prime number checker for numbers between 1 and 250,000." This is what that library looks like:

```
The prime string is a string equal to "NYYNYNYNNNYNYN…"

To decide if a number is prime (fast check):
If the number is less than 1, say no.
If the number is greater than the prime string's length, say no.
Put the prime string's first into a byte pointer.
```

```
Add the number to the byte pointer.
Subtract 1 from the byte pointer.
If the byte pointer's target is the big-Y byte, say yes.
Say no.
```

The prime string in the source is, of course, much longer than the one shown above. And it could be 8 times shorter if we used bits instead of "Y"s to mark the primes. But it hurts my head to have to think that hard. At any rate, this program…

```
To run:
Start up.
Write "Working…" on the console.
Start a timer.
Get a count of prime numbers less than or equal to 250000.
Stop the timer.
Write the count then " prime numbers." on the console.
Write the timer's string then " milliseconds." on the console.
Wait for the escape key.
Shut down.

To get a count of prime numbers less than or equal to a number:
Privatize the number.
Clear the count.
Loop.
If the number is prime (fast check), add 1 to the count.
Subtract 1 from the number.
If the number is less than 2, break.
Repeat.
```

…which is very *clear,* and reasonably *small* (about 400k), is also remarkably *fast*. It can count the 22,044 primes less than or equal to 250,000 in 16 milliseconds or less. And, since it's (indirectly) based on the Sieve of Eratosthenes, it meets the requirements of the original specification!

I think Captain Kirk would approve.