



A Tale of Three Programs

Gerard J. Holzmann

SOME SOFTWARE COMPANIES put prospective employees in front of a whiteboard and challenge them to solve a few programming problems on the fly. A sample problem could be to find a way to sort a billion numbers when you can store only a million in memory. Another could be to give an algorithm that can factor a given number into primes or generate prime numbers up to some preset bound. Quite a few websites describe such problems; a favorite seems to be projecteuler.net.

If you're the candidate, I bet you'll try hard to avoid reproducing the most obvious solution to a problem and instead show some spunk by coming up with something a little more creative. I'll take the problem of generating a sequence of prime numbers as a simple example, just to show how trying to be creative can trip you up.

There's quite a bit of history about creating good methods for generating primes that I'll conveniently skip here. When you're standing in front of the whiteboard, you don't get to Google your answer, anyway.

The simplest way you can probably think of is to walk through every positive integer, up to the requested maximum, and check whether it's divisible by a number other than one and itself. That's a slow, tedious process, so the

game is now to find ways to speed that up. For candidate prime numbers larger than two, there's of course no point in trying numbers that are divisible by two because they can't possibly be prime. With the same reasoning, you also don't have to consider numbers that are divisible by any other prime number you already know about.

This will lead you quickly to some version of the classic algorithm that has been known since the days of the Greek mathematician Eratosthenes (276–194 BC), even if you can't remember his name. You can start drafting your whiteboard algorithm once you realize that if you know the primes up to some number N , the next prime number in the series must be the first number larger than N that's not divisible by any of the primes smaller than N .

An Iterative Version

We can code this in any reasonable programming language, and in quite a few unreasonable ones as well. It can be fun to express the algorithm in Python, C, C++, Scala, or Go, or even in scripting languages such as Tcl or Awk. And, yes, I confess that I've tried most of these, including Awk. Each language offers different features that can simplify the job or make it more interesting. For our current purpose, it'll suffice to stick to just plain old C.

```

int nprimes=1;
int primes[N];
int
main(void) // iterative
{ int i, cand = 3;
  primes[0] = 2;
  printf("2\n");
  while (nprimes < N)
  { for (i = 0; i < nprimes; i++)
    { if (cand % primes[i] == 0)
      { break; // not prime
    }
    if (i == nprimes)
    { primes[nprimes++] = cand;
      printf("%d\n", cand);
    }
    cand += 2;
  }
  return 0;
}

```

FIGURE 1. An iterative algorithm for generating prime numbers. This isn't the best algorithm you could write, but it's a starting point.

In a first attempt, we might come up with the version in Figure 1, which assumes that we provide the value of N in some other way—for example, with a macro directive to the C compiler.

Let's take the first prime number, two, for granted, and try to generate the rest. We consider only odd numbers and eliminate any new number that's divisible by one of the primes discovered before. Clearly this isn't the best we could do, but it's a starting point.

A couple of improvements will quickly come to mind. For example, we could shortcut the search in the for loop and declare a new prime once the value of `primes[i]` starts exceeding the square root of `cand`. We can avoid the potentially costly

```

int nprimes=1;
int primes[N];
int
divides(int cand, int n)
{
  if (n > 0 && divides(cand, n-1))
  { return 1;
  }
  return (cand%primes[n] == 0);
}
int
main(void) // recursive
{ int cand = 3;
  primes[0] = 2;
  printf("2\n");
  while (nprimes < N)
  { if (!divides(cand, nprimes-1))
    { primes[nprimes++] = cand;
      printf("%d\n", cand);
    }
    cand += 2;
  }
  return 0;
}

```

FIGURE 2. A recursive algorithm for generating prime numbers. This example isn't any shorter than the one in Figure 1 but is slightly more interesting.

square root computation by instead checking whether `primes[i]*primes[i] > cand` holds. And, we can even precompute and store the value of those products of primes so that we don't have to compute them over and over again. Those changes should improve the efficiency, but they'll also obfuscate the code a bit. So, let's ignore them for now and look at other alternatives that can make the algorithm more interesting.

Recursion

You might get a gut feeling that a recursive procedure is hiding in our first cut at the algorithm. Didn't somebody once say, "To iterate

is human, to recurse, divine"? It doesn't take much to come up with the recursive version of the code in Figure 2, which isn't any shorter but perhaps is at least a bit less stale.

The Sieve

Neither version of the code we've been discussing is a faithful rendering of what's called the *sieve of Eratosthenes*. Over 2,000 years ago, Eratosthenes also considered the problem of generating primes up to some number N , although not likely in a job interview. He noticed that for any new prime we discover, starting with two, we can eliminate from consideration all multiples of that number as well. So, we can indeed eliminate not only all even numbers but also all multiples of 3, 5, 7, and so on. In this way, we can eliminate more and more numbers with each new prime we discover.

This elimination process should speed up the checks because we have fewer and fewer numbers to check as candidate primes, while we're discovering more and more primes. To store the additional information that records whether a number is worth checking for primeness, we need to store only a single bit of information per number up to our preset bound N . We can use the bit to record whether a number has already been eliminated as a multiple of an earlier prime. This leads to the version of the code in Figure 3, which is now definitely more interesting.

For simplicity, this example uses a byte instead of a bit to record the primeness of each candidate number, but you get the idea. The code is now not only closer to the true sieve algorithm but also a bit shorter. It would be even shorter if we could eliminate the four lines for the final loop over

```

char notprime[MAX];
int
main(void) // sieve
{ int i, cand;
  for (cand = 2; cand < MAX;
      cand++)
    { if (!notprime[cand])
      for (i = cand+1; i < MAX; i++)
        { if (!notprime[i]
            && i % cand == 0)
          { notprime[i] = 1;
            }
        }
      }
    for (i = 2; i < MAX; i++)
      { if (!notprime[i])
        { printf("%d\n", i);
          }
        }
    return 0;
}

```

FIGURE 3. An algorithm for generating prime numbers that's based on the sieve of Eratosthenes. This algorithm is shorter than the ones in Figures 1 and 2.

the numbers, which only serves to print each final prime number once we've reached the preset bound. This last version might be the most pleasing, especially if you can come up with it during your whiteboard session. Is it also the most efficient?

Measuring

Figure 4 shows the result of a quick performance comparison that used each version of the algorithm to generate the first N prime numbers. For the sieve version, that means allocating an array larger than N to ensure that it contains the required N primes. For example, to make sure we can find each of the first 1,000 prime numbers, the `notprime` array from the sieve algorithm must use a definition of `MAX` of at least 7,920, because the 1,000th prime is 7,919.

As we might expect, both peak

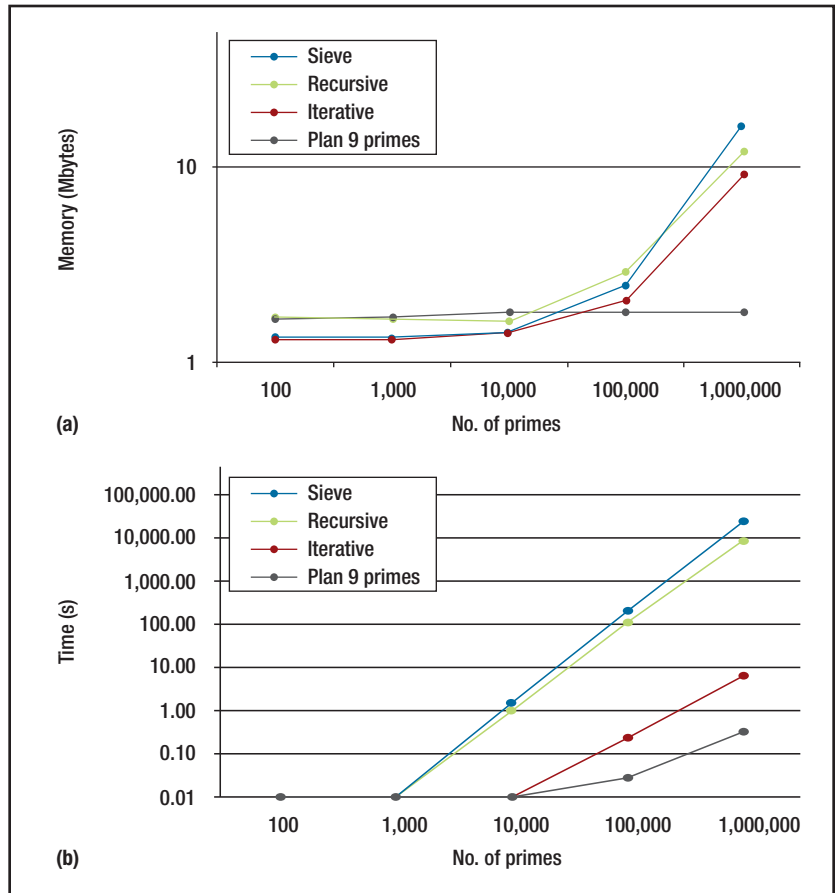


FIGURE 4. The relative performance of a few trial algorithms for computing primes, comparing (a) memory use and (b) runtime. The top three curves show the performance of the three algorithms discussed here, using iteration, recursion, or the prime sieve method. The bottom curve shows the performance of the *primes* command of the Plan 9 OS, which dates back to an early version of Unix from 1975.

memory use and runtime increase as the number of generated primes increase. Figure 4 shows a log-log plot for a sequence of tests I used to generate up to one million primes. In both graphs, lower means better. No significant differences in performance exist between the algorithms up to about 1,000 primes, but beyond that, the differences are stark.

The best-performing algorithm of the three versions we discussed turns out to be the basic one we wrote first, before we started “improving”

it. And yes, each new improvement seems to have worsened the performance a little. The difference in runtime between the best and worst version of the code when generating a million primes is no less than five orders of magnitude. Now, I don't want to be petty about small differences, but the difference between waiting six hours or six seconds for a result is often quite noticeable.

You likely know Donald Knuth's statement that “premature optimization is the root of all evil.”¹ So,

is there really nothing we could do to improve our code's performance beyond that of the first version? Of course there is, but it requires using a little more theory and likely better coding skills than I can bring to the job, as illustrated by the bottom curve in the two graphs in Figure 4.

Morris's Code

The measurements that produced those bottom curves were from running the same tests on the version of the `primes` command I found

in the Plan 9 OS. The code for this command can be traced back to Sixth Edition Unix from 1975. It was initially written in PDP-11 assembly code by Robert Morris Sr. (1932–2011), a mathematician and an original member of the former Unix group at Bell Labs Research. In 1986, Morris left Bell Labs to join the US National Security Agency as the chief scientist of its National Computer Security Center. His `primes` program was rewritten into C shortly before Eighth Edition Unix

was released in 1985; it handles primes up to 2^{56} (about 7.2×10^{16}).

The Plan 9 and Unix version of the `primes` command is still a bit faster than the more recent comparable command, `matho-primes`, that you can install on the Ubuntu OS. It improves over our trial version of the code by another order of magnitude, cutting the time for generating a million primes down to just 0.6 s.

The Plan 9 and Unix versions are approximately 100 lines of C, and the Sixth Edition assembly code version was approximately 350 lines of code. So, with these, we're decidedly no longer in the territory of something that ordinary mortals can produce extemporaneously on a whiteboard.

I find it sobering that the difference in performance between Morris's code and the initial version of the algorithm isn't as large as the degree of harm we inflicted upon ourselves by trying to rewrite that initial version without first measuring its performance. Who among us hasn't fallen into that trap a few times too many? ☹

Reference

1. D. Knuth, "Structured Programming with go to Statements," *Computing Surveys*, vol. 6, no. 4, 1974, pp. 262–301.

GERARD J. HOLZMANN works on developing stronger methods for the design and analysis of safety-critical software as a consultant and researcher at Nimble Research. Contact him at gholzmann@acm.org.

myCS Read your subscriptions through the myCS publications portal at <http://mycs.computer.org>

CONFERENCES
in the Palm of Your Hand

IEEE Computer Society's Conference Publishing Services (CPS) is now offering conference program mobile apps! Let your attendees have their conference schedule, conference information, and paper listings in the palm of their hands.

The conference program mobile app works for Android devices, iPhone, iPad, and the Kindle Fire.

For more information please contact cps@computer.org

IEEE IEEE Computer Society CPS