The Genuine Sieve of Eratosthenes

Melissa E. O'Neill

Harvey Mudd College, Claremont, CA, U.S.A. (e-mail: `oneill@acm.org`)

Abstract

A much beloved and widely used example showing the elegance and simplicity of lazy functional programming represents itself as "The Sieve of Eratosthenes". This paper shows that this example is *not* the sieve, and presents an implementation that actually is.

1 Introduction

The Sieve of Eratosthenes is a beautiful algorithm that has been cited in introductions to lazy functional programming for more than thirty years (Turner, 1975). The Haskell code below is fairly typical of what is usually given:

```
primes = sieve [2..]

sieve (p : xs) = p : sieve [x | x <- xs, x 'mod' p > 0]
```

The code is short, looks elegant, and seems to make a persuasive case for the power of lazy functional programming. Unfortunately, on closer inspection, that case begins to fall apart. For example, the above algorithm actually runs rather slowly, sometimes inspiring excuses as extreme as this one:

Try primes !! 19. *You should get* 71. (*This computation may take a few seconds, and do several garbage collections, as there is a lot of recursion going on.*)[1]

A skeptic might very well ask whether it is really okay for computing the first few thousand primes (or, in the above case, only first twenty!) to be such a taxing problem, and begin to wonder whether laziness or functional programming as a whole is a hopelessly inefficient waste of time.

The culprit, however, is neither laziness nor functional programming: It is the algorithm. Despite widespread assertion to the contrary, *this algorithm is **not** the Sieve of Eratosthenes!*

This paper shows

- Why this widely-seen implementation is *not* the Sieve of Eratosthenes;
- How an algorithm that *is* the Sieve of Eratosthenes may be written in a lazy functional style; and
- How our choice of data structure matters.

---

[1] This rather extreme example was found in a spring, 2006, undergraduate programming-languages assignment used by several well-respected universities. The original example was not in Haskell (where typical systems require a few orders of magnitude more primes before they bog down), but I have modified it to use Haskell syntax to fit with the rest of this paper.

In passing, we will look at the time complexity of the prime-finding methods we examine, and in doing so discover how we may analyze their performance in a fairly straightforward way.

2 What the Sieve Is and Is Not

Let us first describe the original "by hand" sieve algorithm as practiced by Eratosthenes. We start with a table of numbers (e.g., 2, 3, 4, 5, . . .) and progressively cross off numbers in the table until the only numbers left are primes. Specifically, we begin with the first number, p, in the table, and

1. Declare p to be prime, and cross off all the multiples of that number in the table, starting from p^2;
2. Find the next number in the table after p that is not yet crossed off and set p to that number; and then repeat from step 1.

The starting point of p^2 is a pleasing but minor optimization, which can be made because lower multiples will have already been crossed off when we found the primes prior to p. For a fixed-size table of size n, once we have reached the $\sqrt{n}^{\text{th}}$ entry in the table, we need perform no more crossings off—we can simply read the remaining table entries and know them all to be prime. (This optimization does not affect the time complexity of the sieve, however, so its absence from the code in Section 1 is not our cause for worry.)

The details of what gets crossed off, when, and how many times, are key to the efficiency of Eratosthenes algorithm. For example, suppose that we are finding the first 100 primes (i.e., 2 through 541), and have just discovered that 17 is prime, and need to "cross off all the multiples of 17". Let us examine how Eratosthenes's algorithm would do so, and then how the algorithm from Section 1 would do so.

In Eratosthenes's algorithm, we start crossing off multiples of 17 at 289 (i.e., 17×17) and cross off the multiples $289, 306, 323, \ldots, 510, 527$, making fifteen crossings off in total. Notice that we cross off 306 (17×18), even though it is a multiple of both 2 and 3 and has thus already been crossed off twice.[2] The algorithm is efficient because each composite number, c, gets crossed off f times, where f is the number of unique factors of c less than $\sqrt{c}$. The average value for f increases slowly, being less than 3 for the first 10^{12} composites, and less than 4 for the first 10^{34}.[3]

Contrast the above behavior with that of the algorithm from Section 1, which I shall call "the unfaithful sieve". After finding that 17 is prime, the unfaithful sieve will check all the numbers not divisible by $2, 3, 5, 7, 11$ or 13 for divisibility by 17. It will perform this test on a total of ninety-nine numbers $(19, 23, 29, 31, \ldots, 523, 527)$.

The difference between the two algorithms is not merely that the unfaithful sieve doesn't perform "optimizations", such as starting at the square of the prime, or

---

[2] Optimizations to the Sieve of Eratosthenes usually special-case the first prime, 2, and sometimes other small primes. We'll discuss those optimizations in Section 3.2.

[3] The number of unique factors in a number n is usually written $\omega(n)$, and $\omega(n) \approx \ln \ln n$ (Hardy & Wright, 1979). Thus the average value for f is $\Theta(\log \log n)$.

that it uses a divisibility check rather than using a simple increment. For example, even if it did (somehow) begin at 289, it would still check all forty-five numbers that are not multiples of the primes prior to 17 for divisibility by 17 (i.e., $289, 293, 307, \ldots, 523, 527$). At a fundamental level, these two algorithms "cross off all the multiples of 17" differently.

In general, the speed of the unfaithful sieve depends on the number of primes it tries that are *not* factors of each number it examines, whereas the speed of Eratosthenes's algorithm depends on the number of (unique) primes that *are*. We will discuss how this difference impacts their time complexity in the next section.

Some readers may feel that despite all of these concerns, the earlier algorithm is somehow "morally" the Sieve of Eratosthenes. I would argue, however, that they are confusing a mathematical abstraction drawn from the Sieve of Eratosthenes with the actual algorithm. The algorithmic details, such as *how* you remove all the multiples of 17, *matter*.

If this algorithm is not the Sieve of Eratosthenes, what is it? In fact it is a simple naïve algorithm, known as *trial division*, that checks the primality of x by testing its divisibility by each of the primes less than x. But even this naïve algorithm would normally be more efficient, because we would typically check only the primes up to $\sqrt{x}$. We can write trial division more clearly as

```
primes     = 2 : [x | x <- [3..], isprime x]
isprime x  = all (\p -> x 'mod' p > 0) (factorsToTry x)
  where
    factorsToTry x  = takeWhile (\p -> p*p <= x) primes
```

2.1 Performance in Theory and Practice

To futher convince ourselves that we are are not looking at the same algorithm, and to further understand why it matters, it is useful to look at the time performance of the algorithms we have examined so far, both in theory and in practice. For asymptotic time performance, we will examine the time it takes to find all the primes less than or equal to n.

The Sieve of Eratosthenes implemented in the usual way requires $\Theta(n \log \log n)$ operations to find all the primes up to n. This result is widely known, but we can derive this result for ourselves by noting that we perform n/p crossings off for each prime p, and thus the performance of the sieve is

$$\sum_{i=1}^{\pi(\sqrt{n})} \frac{n}{p_i} \approx \frac{n}{2} + n \sum_{i=2}^{\frac{2\sqrt{n}}{\ln n}} \frac{1}{i \ln i} \approx \frac{n}{2} + n \int_{i=2}^{\frac{2\sqrt{n}}{\ln n}} \frac{1}{i \ln i} \approx n \ln \ln n + O(n)$$

where, from the Prime Number Theorem, $\pi(x) \approx x/\ln x$ is the number of primes less than x, and $p_i \approx i \ln i$ is the ith prime.[4] (When we apply this approximation,

---

[4] If we start crossing off at p^2 rather than p, the number of composites we cross off is $n/p - p + 1$, but it makes no significant difference to our sum, because it only subtracts an irrelevant $O(n/\log n)$ factor.

we separate out the first prime to avoid the terrible approximation that the first prime is zero).

Let us now turn our attention to trial division. Although we can use industrial-strength number theory (see Section 5) to derive a bound, instead we will take a slightly gentler and hopefully more accessible route. We will create a tight bound by underestimating the amount of work the algorithm does to provide a lower bound and then overestimating it to generate an upper bound, and then we shall find that both bounds are the same, asymptotically.

First, let us underestimate by considering only the work the algorithm does on primes, ignoring all composites. Each prime will be divided by all primes less than its square root (the first two primes, 2 and 3, won't be divided by any primes since there are no primes less than their square root). Thus, the number of attempted divisions is

$$\sum_{i=3}^{\pi(n)} \pi(\sqrt{p_i}) \approx 2 \sum_{i=3}^{\frac{n}{\ln n}} \frac{\sqrt{i \ln i}}{\ln(\sqrt{i \ln i})} \approx 2 \int_3^{\frac{n}{\ln n}} \frac{\sqrt{i \ln i}}{\ln(i \ln i)} \mathrm{d}i \approx \frac{4}{3} \frac{n\sqrt{n}}{(\ln n)^2}$$

Having found a lower bound, let us now create an upper bound by also considering the work required for the composites. We can actually afford to be generous and *overcount* the work done processing composites as follows: all multiples of two (all $n/2$ of them) are handled with one trial division, but let us go on to assume that all multiples of three ($n/3$ of them) require two trial divisions, all multiples of five ($n/5$ of them) require three trial divisions, and so forth. Clearly we are overcounting because some of the multiples of three are also multiples of two and are thus counted twice, and likewise most of the multiples of five are counted two or three times! In general, the amount of work done in processing the composites, with considerable overcounting, is

$$\sum_{i=1}^{\pi(\sqrt{n})} i \frac{n}{p_i} \approx \frac{n}{2} + n \sum_{i=2}^{\frac{2\sqrt{n}}{\ln n}} \frac{i}{i \ln i} \approx \frac{n}{2} + n \int_{i=2}^{\frac{2\sqrt{n}}{\ln n}} \frac{1}{\ln i} \approx \frac{2n\sqrt{n}}{(\ln n)^2} + \mathrm{O}(n)$$

Including the work done in processing noncomposites (primes), which we derived earlier, an upper bound on the number of trial divisions is $\frac{10}{3} n\sqrt{n}/(\ln n)^2$. From these upper and lower bounds, we can conclude that trial division has time complexity $\Theta(n\sqrt{n}/(\log n)^2)$.

The unfaithful sieve does the same amount of work on the composites as normal trial division (because on a composite i, one of the primes $< \sqrt{i}$ will divide it), but it tries to divide primes by *all* prior primes. Thus, the amount of work done on the primes is

$$\sum_{i=1}^{\pi(n)} (i - 1) = \frac{1}{2}\pi(n)(\pi(n) - 1) \approx \frac{n^2}{2(\ln n)^2}$$

and thus the unfaithful sieve has time complexity $\Theta(n^2/(\log n)^2)$.

Thus, we can see that from a time-complexity standpoint, the unfaithful sieve is asymptotically worse than simple trial division, and that in turn is asymptotically worse than than the true Sieve of Eratosthenes.
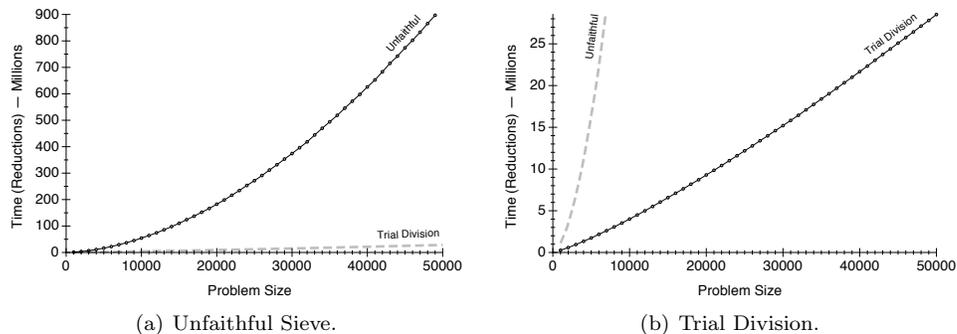
(a) Unfaithful Sieve.   (b) Trial Division.

Fig. 1. Time Performance of The Unfaithful Sieve and Trial Division[5]

The performance results shown in Figure 1 provide some practical support for our theoretical results, showing the performance of trial division and the unfaithful sieve finding all the primes less than some n. We can see that not only are the asymptotics of trial division better than those of the unfaithful sieve, but the constant factors are also dramatically improved.

To draw a similar graph for the Sieve of Eratosthenes, we require a functional implementation of the algorithm, which we discuss next.

3 An Incremental Functional Sieve

Despite their other drawbacks, the implementations of the unfaithful sieve and trial division that we have discussed use functional data structures and produce an infinite list of primes. In contrast, classic imperative implementations of the Sieve of Eratosthenes use an array and find primes up to some fixed limit. Can the genuine Sieve of Eratosthenes also be implemented efficiently and elegantly in a purely functional language and produce an infinite list? Yes!

Whereas the original algorithm crosses off all multiples of a prime at once, we perform these "crossings off" in a lazier way: crossing off just-in-time. For this purpose, we will store a table in which, for each prime p that we have discovered so far, there is an "iterator" holding the next multiple of p to cross off. Thus, instead of crossing off all the multiples of, say, 17, at once (impossible, since there are infinitely many for our limit-free algorithm), we will store the first one (at 17×17; i.e., 289) in our table of upcoming composite numbers. When we come to consider whether 289 is prime, we will check our composites table and discover that it is a known composite with 17 as a factor, remove 289 from the table, and insert 306 (i.e., $289+17$). In essence, we are storing "iterators" in a table keyed by the current value of each iterator.

Let us consider what kind of data structure would be useful for storing our table.

---

[5] I have followed Runciman (1997) in measuring time performance according to the number of reduction steps performed by the *Hugs* Haskell interpreter. Similar time results can be obtained using a Haskell compiler, such as *GHC*.

Just as the standard Sieve of Eratosthenes algorithm would cross off 306 three times (for 2, 3, and 17), so our algorithm will eventually move the iterators for 17, 3, and 2 so that their current positions are all 306, and then, when we hit 306, these iterators will separate and go on their separate ways. Given our need for insert and lookup in the table, and for multiple iterators to pass over the same point, perhaps a multimap seems appropriate. In Haskell, Data.Map provides all the necessary operations (insertWith (++) merges a list to be inserted with any list already stored at that entry). We can thus code sieve as follows:

```
sieve xs = sieve' xs Map.empty
  where
    sieve' []      table = []
    sieve' (x:xs) table =
        case Map.lookup x table of
            Nothing      -> x : sieve' xs (Map.insert (x*x) [x] table)
            Just facts   -> sieve' xs (foldl reinsert (Map.delete x table) facts)
          where
            reinsert table prime = Map.insertWith (++) (x+prime) [prime] table
```

The time complexity of this algorithm appears slightly worse than typical imperative renditions of the sieve because of the $\Theta(\log n)$ cost of using a a tree, thus the performance for finding all primes less than n is $\Theta(n \log n \log \log n)$, which is nevertheless better than the $\Theta(n\sqrt{n}/(\log n)^2)$ performance of trial division or the $\Theta(n^2/(\log n)^2)$ performance of the unfaithful sieve. Moreover, if we assume that arithmetic operations themselves have non-unit cost, all the algorithms require at least a $\log n$ additional time factor, which can offset the cost of the tree (with a cleverly structured tree, such as a Braun tree or binary trie (Paulson, 1996), tree accesses would not suffer this extra $\log n$ overhead). Thus, if we were concerned about the complexity overheads of using a tree, we could easily address those concerns.

Figure 2(a) shows the time performance of this sieve implementation finding the ith prime; the performance is much improved over that of the unfaithful sieve algorithm. Despite being asymptotically better than trial division, the constant factors are much higher—if you're calculating more than about a million primes, the sieve wins out. To definitively trounce trial division, we should look at bringing those constant factors down a little.

3.1 Using a Better Data Structure

A multimap is actually overkill for this problem, because we only examine the table looking for composites in increasing order. Thus we only need to check whether a candidate prime is the least element in the table (thereby finding it to be composite) or find that the least element in the table is greater than our candidate prime, revealing that our candidate actually is prime. Given these needs, a priority queue is an attractive choice, especially since this data structure natively supports multiple items with the same priority (dequeuing in them arbitrary order).

Haskell does not provide a built-in priority queue type, but heap-based functional

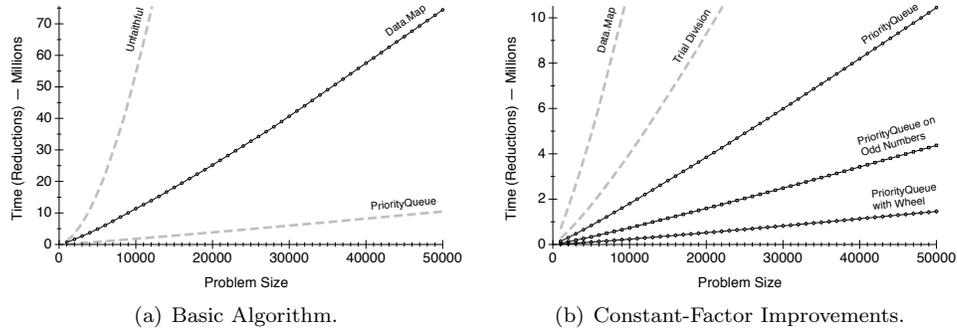(a) Basic Algorithm.　　　　　　(b) Constant-Factor Improvements.

Fig. 2. Time Performance of the Actual Sieve of Eratosthenes

implementations are easy enough to find (Paulson, 1996). We will suppose a priority
queue type that includes the operations

```
empty              :: PriorityQ k v
minKey             :: PriorityQ k v –> k
minKeyValue        :: PriorityQ k v –> (k,v)
insert             :: Ord k => k –> v –> PriorityQ k v –> PriorityQ k v
deleteMinAndInsert :: Ord k => k –> v –> PriorityQ k v –> PriorityQ k v
```

(We provide deleteMinAndInsert because a heap-based implementation can support
this operation with considerably less rearrangement than a deleteMin followed by an
insert.)

We can adjust our previous sieve code to use to use the priority queue as follows:

```
sieve [] = []
sieve (x:xs) = x : sieve' xs (insertprime x xs PQ.empty)
  where
    insertprime p xs table = PQ.insert (p*p) (map (* p) xs) table
    sieve' []      table = []
    sieve' (x:xs) table
      | nextComposite <= x    = sieve' xs (adjust table)
      | otherwise             = x : sieve' xs (insertprime x xs table)
      where
        nextComposite = PQ.minKey table
        adjust table
          | n <= x     = adjust (PQ.deleteMinAndInsert n' ns table)
          | otherwise  = table
        where
          (n, n':ns) = PQ.minKeyValue table
```

Also, rather than represent our "iterators" as a simple increment, our table now
stores lazy lists, giving our iterators more flexibility. The iterator corresponding to
a prime p is a lazy list formed by multiplying our current input list by p. If the
input list advances by one, the iterator for a prime p advances by p as before, but

if the input list advances by twos, as happens below, our iterator will advance by $2 \times p$ as we would desire. Thus we can cut down the work for our sieve by skipping the even numbers as follows:

```
primes = 2 : sieve [3,5..]
```

At this point, the benefits of a functional approach are becoming more apparent. Most implementations of the Sieve of Eratosthenes avoid checking multiples of two, but do so by judiciously adding a few 2*x and x+2s to the code. This implementation merely requires a change to its input data.

Figure 2(b) shows the time performance of this improved implementation. The top line shows the time to find the nth prime using the sieve applied to [2..], whereas the second line uses the sieve on odd numbers (the bottom line is discussed in Section 3.2). As we can see, using this optimization, we cut our execution time by a factor of about three.

3.2 Using a Simple Wheel

Why stop at eliminating multiples of 2? More than a third of our composites are divisible by 3, and more than a fifth of them are divisible by 5. If, say, we avoid seeing multiples of 2, 3, 5, and 7, we can eliminate more than 77% of our work for large n and even more for smaller n—in general, in the range 2–n, there are $n - \pi(n) - 1$ composites, and about $n \prod_{k=1}^{i} (1 - \frac{1}{p_k}) - \pi(n) - 1$ composites are not divisible by the first i primes.

As we saw above, to produce numbers that are not multiples of 2, we simply begin at 3 and then keep adding 2. To avoid multiples of both 2 and 3, we can begin at 5 and alternately add 2 then 4. We can visualize this technique as a wheel of circumference 6 with holes at a distance of 2 and 4 rolling up the number line. In general, adding an additional prime p to the wheel multiplies the circumference of the wheel by p, and removes every p^{th} composite. Thus, there are usually diminishing returns for large wheel sizes—our wheel for the first four primes has circumference 210 (i.e., $2 \times 3 \times 5 \times 7$) with 48 holes, whereas the wheel for eight primes has circumference 9,699,690 and 1,658,880 holes, but eliminates fewer than 7% of the remaining composites.

We can easily hardcode a small wheel and use it as follows:[6]

```
wheel2357    = 2:4:2:4:6:2:6:4:2:4:6:6:2:6:4:2:6:4:6:8:4:2:4:2:4:8
                :6:4:6:2:4:6:2:6:6:4:2:4:6:2:6:4:2:4:2:10:2:10:wheel2357

spin (x:xs) n  = n : spin xs (n + x)

primes          = 2 : 3 : 5 : 7 : sieve (spin wheel2357 11)
```

Thus we can see that the common optimizations made to the sieve easily apply

---

to our algorithm. In fact, the usual C language implementation of the sieve does not support this optimization nearly as easily—changes to the core sieve algorithm are usually required.

The lowest line in Figure 2(b) shows the performance of the algorithm sieving numbers generated by the 2-3-5-7 wheel. We can see that this version runs more than seven times faster than sieving all the composites, and three times faster than sieving odd numbers.

Interestingly, using a wheel to provide the input for the unfaithful-sieve algorithm makes little difference to its performance. For very small n there is some benefit, but that benefit diminishes quickly as n increases, with less than a 4.5% time reduction for $n > 5000$, and less than 1% for $n > 36000$. In fact, were we to draw a graph for the performance of the unfaithful sieve with the wheel optimization, it would be visually indistinguishable from the one in Figure 1(a). From Section 2.1, we know that the poor performance of the unfaithful sieve is due entirely to the huge amount of work it expends on primes, trying to divide each new prime, p, by every previously discovered prime (all $\pi(p) - 1$ of them); as n increases there is little difference between attempting $\pi(n) - 1$ divisions without the 2-3-5-7 wheel and $\pi(n) - 5$ divisions with it.

4 Conclusion

A "one liner" to find a lazy list of prime numbers *is* a compelling example of the power of laziness and the brevity that can be achieved with the powerful abstractions present in functional languages. But, despite fooling some of us for years, the algorithm we began with isn't the real sieve, nor is it even the most efficient one liner that we can write.

An implementation of the actual sieve has its own elegance, showing the utility of well-known data structures over the simplicity of lists. It also provides a compelling example of why data structures such as heaps exist even when other data structures have similar $O(\log n)$ time complexity—choosing the right data structure for the problem at hand made an order of magnitude performance difference.

The unfaithful-sieve algorithm does have a place as an example. It is very short, and it also serves as a good example of how elegance and simplicity can beguile us. Although the name *The Unfaithful Sieve* has a certain ring to it, given that the unfaithful algorithm is nearly a thousand times slower than our final version of the real thing to find about 5000 primes, we should perhaps call it *The Sleight on Eratosthenes*.

5 Related Work

Runciman (1997) suggests several enhancements to the basic unfaithful-sieve algorithm, which he claims, as so many others have, to be the Sieve of Eratosthenes. His enhancements do improve the running time dramatically, but the empirical evidence from running his code seems to show that his changes merely affect the constant

factors in the approach, not its asymptotics. Thus, alas, at about the 125,000[th] prime, his algorithm is overtaken by simple trial division.

Meertens (2004) presents some ways to make the behavior of the unfaithful sieve easier for newcomers to functional programming to understand, but also improperly describes it as being the Sieve of Eratosthenes.

The Sieve of Eratosthenes is a classic algorithm. It has inspired and been surpassed by more powerful algorithms and helped to spawn a rich literature (e.g., Pritchard, 1987). But while the current state of the art in prime-number sieves is interesting, current techniques are largely tangential to the purposes of this paper, which was merely to reproduce the original Sieve of Eratosthenes as faithfully as possible in a lazy functional form. Incremental algorithms exist for performing sieving (Bengelloun, 1986; Pritchard, 1994), but these algorithms are typically array-based. Crandall and Pomerance (2001) provide a good primer on the state of prime computation in general.

Trial division is most often used not as an algorithm for finding primes, but as a factoring algorithm. In that context, we do not stop when we successfully divide the number, we keep dividing to find all of its factors, with time complexity $\Theta(\sqrt{n}/\log n)$ for each integer we factor (c.f., the $\Theta(\sqrt{n}/(\log n)^2)$ per integer examine we found for our application of trial division, where we stop at the first divisor we find).

Those wishing to use "industrial-strength number theory" to bound the performance of trial division and the unfaithful sieve, instead of the techniques we used in Section 2.1, will find the Φ function useful. $\Phi(n, p)$ is the number of integers $\leq n$ not divisible by any prime $< p$. Using this function, the performance of the unfaithful sieve is simply $\sum_{i=2}^{\pi(n)} \Phi(n, p_i)$ and that of trial division is $\sum_{i=2}^{\pi(\sqrt{n})} \Phi(n, p_i)$. Selberg (1946) showed that $\Phi(n, p) \in \Theta(n/\log p)$. Φ is closely related to Legendre's Formula and Merten's theorem.

6 Epilogue

In discussing earlier drafts of this paper with other members of the functional programming community, I discovered that some functional programmers prefer to work solely with lists whenever possible, despite the ease with which languages such as Haskell and ML represent more advanced data structures. Thus a frequent question from readers of earlier drafts whether a genuine Sieve of Eratosthenes could be implemented using only lists. Some of those readers wrote their own implementations to show that you can indeed to so.

In a personal communication, Richard Bird suggested the following as a faithful list-based implementation of the Sieve of Eratosthenes. This implementation maps well to the key ideas of this paper, so with his permission I have reproduced it. The composites structure is our "table of iterators", but rather than using a tree or heap to represent the table, he uses a simple list of lists. Each of the inner lazy lists corresponds to our "iterators". Removing elements from the front of the union of this list corresponds to removing elements from our priority queue.

```
primes  =  2:([3..] 'minus' composites)
  where
    composites  =  union [multiples p | p <- primes]

multiples n  =  map (n*) [n..]

(x:xs) 'minus' (y:ys)   | x < y   = x:(xs 'minus' (y:ys))
                        | x == y  = xs 'minus' ys
                        | x > y   = (x:xs) 'minus' ys

union  =  foldr merge []
  where
    merge (x:xs) ys = x:merge' xs ys
    merge' (x:xs) (y:ys)   | x < y   = x:merge' xs (y:ys)
                           | x == y  = x:merge' xs ys
                           | x > y   = y:merge' (x:xs) ys
```

This code makes careful use of laziness. In particular, Bird remarks that "Taking the union of the infinite list of infinite lists [[4,6,8,10,..], [9,12,15,18..], [25,30,35,40,...],...] is tricky *unless* we exploit the fact that the first element of the result is the first element of the first infinite list. That is why union is defined in the way it is in order to be a productive function."

While this incarnation of the Sieve of Eratosthenes does achieve the same ends as our earlier implementations, its list-based implementation does not give the same asymptotic performance. The structure of Bird's table, in which the list of composites generated by the kth prime is the kth element in the outer list, means that when we are checking the ith number for primality, union requires $\sum_{k=1}^{\pi(\sqrt{i})} k/p_k \in \Theta(\sqrt{i}/(\log i)^2)$ time, resulting in a time complexity of $\Theta(n\sqrt{n}\log\log n/(\log n)^2)$, making it asymptotically worse than trial division, but only by a factor of $\log\log n$.

In practice, Bird's version is good enough for many purposes. His code is about four times faster than our trial-division implementation for small n, and because $\log\log n$ grows very slowly, it is faster for all practical sizes of n. It is also faster than our initial tree-based code for $n < 10^{8.5}$, and faster than the basic priority-queue version for $n < 275,000$, but never faster than the priority-queue version that uses the wheel. Incidentally, Bird's algorithm could be modified to support the wheel optimizations, but the changes are nontrivial (in particular, multiples would need to take account of the wheel).

For any problem, there is a certain challenge in trying to solve it elegantly using only lists, but there are nevertheless good reasons to avoid too much of a fixation on lists, particularly if a focus on seeking elegant list-based solutions induces a myopia for elegant solutions that use other well-known data structures. For example, some of the people with whom I discussed the ideas in this paper were not aware that a solution using a heap was possible in a purely functional language because they had never seen one used in a functional context. The vast majority of well-understood standard data structures can be as available in a functional environment as they are in an imperative one, and in my opinion, we should not be afraid to be seen to use them.

7 Acknowledgments

References

Bengelloun, S. A. (1986). An incremental primal sieve. *Acta informatica*, **23**, 119–125.

Crandall, Richard, & Pomerance, Carl. (2001). *Prime numbers. A computational perspective.* New York: Springer-Verlag.

Hardy, G. H., & Wright, E. M. (1979). *An introduction to the theory of numbers.* 5th edn. Clarendon Press. Pages 354–358.

Meertens, Lambert. (2004). Calculating the Sieve of Eratosthenes. *Journal of functional programming*, **14**(6), 759–763.

Paulson, Lawrence C. (1996). *ML for the working programmer.* 2nd edn. Cambridge University Press.

Pritchard, Paul. (1987). Linear prime-number sieves: A family tree. *Science of computer programming*, **9**, 17–35.

Pritchard, Paul. (1994). Improved incremental prime number sieves. *Pages 280–288 of: Proceedings of the first international symposium on algorithmic number theory.* London, UK: Springer-Verlag.

Runciman, Colin. (1997). Lazy wheel sieves and spirals of primes. *Journal of functional programming*, **7**(2), 219–225.

Selberg, Sigmund. (1946). An upper bound for the number of cancelled numbers in the Sieve of Eratosthenes. *Det kongelige Norske videnskabers selskabs forhandlinger, Trondhjem*, **19**(2), 3–6.

Turner, David A. (1975). *SASL language manual.* Tech. rept. CS/75/1. Department of Computational Science, University of St. Andrews.