Programming with Prime Numbers

Contents

Prime numbers are those integers greater than one that are divisible only by themselves and one; an integer greater than one that is not prime is composite. Prime numbers have fascinated mathematicians since the days of the ancient Greek mathematicians, and remain an object of study today. The sequence of prime numbers begins 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, ... and continues to infinity, which Euclid, the famous teacher of geometry, famously proved about twenty-three centuries ago in what must certainly be the most beautiful proof in all of mathematics:

> Assume for the moment that the number of primes is finite, and make a list of them: $p_1, p_2, \ldots, p_k$. Now compute the number $n = p_1 \cdot p_2 \cdot \ldots \cdot p_k + 1$. Certainly n is not evenly divisible by any of the primes p_1, p_2, ..., p_k, because division by any them leaves a remainder of 1. Thus either n is prime, or n is composite but has two or more prime factors not on the list $p_1, p_2, \ldots, p_k$ of prime numbers. In either case the assumption that the number of primes is finite is contradicted, thus proving the infinitude of primes. —Euclid, *Elements*, Book IX, Proposition 20, circa 300 B.C.

In this essay we will examine three problems related to prime numbers: enumerating the prime numbers, determining if a given number is prime or composite, and factoring a composite number into its prime factors. We describe in detail five relevant functions: one that makes a list of the prime numbers less than a given number using the Sieve of Eratosthenes; two that determine whether a given number is prime or composite, one using trial division and the other using an algorithm developed by Gary Miller and Michael Rabin; and two that find the unique factorization of a given composite number, one using trial division and the other using John Pollard's rho algorithm. We first describe the algorithms in the body of the essay, then describe actual implementations in five languages—C, Haskell, Java, Python and Scheme—in a series of appendices.

Although we are primarily interested in programming, we are also careful to honor the underlying mathematics. We have already given Euclid's proof of the infinitude of primes, and we will see his proof of the Fundamental Theorem of Arithmetic. We will also discuss Fermat's Little Theorem, the Prime Number Theorem, the Chinese Remainder Theorem, and the Birthday Paradox. And in addition to Euclid we will meet such great mathematicians as Eratosthenes, Sun Zi, Aryabhata, Fermat, Leibniz, Euler, and Gauss, as well as some contemporary mathematicians.

Our goals are modest. Our purpose is pedagogical, so we are primarily interested in the clarity of the code. We describe algorithms that are well known and implement them carefully. And we hope that careful reading will lead you to be a better programmer in addition to learning something about prime numbers. Even so, our functions are genuinely useful for a variety of purposes beyond simple study, and are actually suitable for real work in the study of primes.

1 The Sieve of Eratosthenes

The method that is in common use today to make a list of the prime numbers less than a given input n was invented about two hundred years before Christ by Eratosthenes of Cyrene, who was an astronomer, geographer and mathematician, as well as the third chief librarian of Ptolemy's Great Library at Alexandria; he calculated the distance from Earth to Sun, the tilt of the Earth's axis, and the circumference of the Earth, and invented the leap day and a system of latitude and longitude. His method begins by making a list of all the numbers from 2 to the desired maximum prime number n. Then the method enters an iterative phase. At each step, the smallest uncrossed number that hasn't yet been considered is identified, and all multiples of that number are crossed out; this is repeated until no uncrossed numbers remain unconsidered. All the remaining uncrossed numbers are prime.

Thus, the first step crosses out all multiples of 2: 4, 6, 8,10 and so on. At the second step, the smallest uncrossed number is 3, and multiples of 3 are crossed out: 6, 9, 12, 15 and so on; note that some numbers, such as 6, might be crossed out multiple times. At this point 4 has been crossed out, so the next smallest uncrossed number is 5, and its multiples 10, 15, 20, 25 and so on are also crossed out. The process continues until all uncrossed numbers have been considered. Thus, each prime is used to "sift" its multiples out of the original list, so that only primes are left in the sieve. Here is a formal statement of the algorithm:

Algorithm 1.A: Ancient Sieve of Eratosthenes: Generate the primes not exceeding $n > 1$:

1. [*Initialization*] Create a bitarray $B_{2\ldots n}$ with each item set to `TRUE`. Set $p \leftarrow 2$.
2. [*Terminate?*] If $p > n$, stop.
3. [*Found prime?*] If $B_p =$ `FALSE`, go to Step 5. Otherwise, output the prime p and set $i \leftarrow p + p$.
4. [*Sift on* p] Set $B_i \leftarrow$ `FALSE`. Set $i \leftarrow i + p$. If $i \leq n$, go to Step 4.
5. [*Iterate*] Set $p \leftarrow p + 1$ and go to Step 2.

Or instead of a formal algorithm you may prefer the ditty from the 1960 book *Drunkard's Walk* by Frederik Pohl:

> Strike the Twos and strike the Threes,
> The Sieve of Eratosthenes!
> When the multiples sublime,
> The numbers that are left, are prime.

Although this is the basic algorithm, there are three optimizations that are routinely applied. First, since 2 is the only even prime, it is best to handle 2 separately and sieve only on odd numbers, reducing the size of the sieve by half. Second, instead of starting the crossing-out at the smallest multiple of the current sieving prime, it is possible to start at the square of the multiple, since all smaller numbers will have already been crossed out; we saw that in the sample when 6 was already crossed out as a multiple of 2 when we were crossing out multiples of 3. Third, as a consequence of the second optimization, sieving can stop as soon as the square of the sieving prime is greater than n, since there is nothing else to do. Here is a formal statement of the algorithm for the optimized Sieve of Eratosthenes:

Algorithm 1.B: Optimized Sieve of Eratosthenes: Generate the primes not exceeding $n > 1$:

1. [*Initialization*] Set $m \leftarrow \lfloor (n-1)/2 \rfloor$. Create a bitarray $B_{0\ldots m-1}$ with each item set to `TRUE`. Set $i \leftarrow 0$. Set $p \leftarrow 3$. Output the prime 2.
2. [*Sieving complete?*] If $n < p^2$, go to Step 5.
3. [*Found prime?*] If $B_i =$ `FALSE`, set $i \leftarrow i + 1$, set $p \leftarrow p + 2$, and go to Step 2. Otherwise, output the prime p and set $j \leftarrow 2i^2 + 6i + 3$ (or $j \leftarrow (p^2 - 3)/2$).
4. [*Sift on* p] If $j < m$, set $B_j \leftarrow$ `FALSE`, set $j \leftarrow j + 2i + 3$ (or $j \leftarrow j + p$) and go to Step 4. Otherwise, set $i \leftarrow i + 1$, set $p \leftarrow p + 2$, and go to Step 2.
5. [*Terminate?*] If $i = m$, stop.
6. [*Report remaining primes.*] If $B_i =$ `TRUE`, output the prime p. Then, regardless of the value of B_i, set $i \leftarrow i + 1$, set $p \leftarrow p + 2$, and go to Step 5.

The calculation of j in Step 3 is interesting. Since the bitarray contains odd numbers starting from 3, an index i of the bitarray corresponds to the number $p = 2i + 3$; for instance, the fifth item in the bitarray, at index 4, is 11. Sifting starts from the square of the current prime, so to sift the prime 11 at index 4 we start from $11^2 = 121$ at index 59, calculated as $(121 - 3)/2$. Thus, to compute the starting index j, we calculate $((2i + 3)^2 - 3)/2$, which with a little bit of algebra simplifies to the formula in Step 3. You may prefer the alternate calculation $(p^2 - 3)/2$ which exploits the identity $2i + 3 = p$.

As an example, we show the calculation of the primes less than a hundred. The first step notes that 3 is the smallest uncrossed number, and crosses out starting from 3^2: 9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99.

3 5 7 ~~9~~ 11 13 ~~15~~ 17 19 ~~21~~ 23 25 ~~27~~ 29 31 ~~33~~ 35
37 ~~39~~ 41 43 ~~45~~ 47 49 ~~51~~ 53 55 ~~57~~ 59 61 ~~63~~ 65 67 ~~69~~
71 73 ~~75~~ 77 79 ~~81~~ 83 85 ~~87~~ 89 91 ~~93~~ 95 97 ~~99~~

Now 5 is the next smallest uncrossed number, so we cross out starting from 5^2: 25, 35, 45, 55, 65, 75, 85, 95. Note that 45 and 75 were previously crossed out, so only six additional numbers are now crossed.

3 5 7 ~~9~~ 11 13 ~~15~~ 17 19 ~~21~~ 23 ~~25~~ ~~27~~ 29 31 ~~33~~ ~~35~~
37 ~~39~~ 41 43 ~~45~~ 47 49 ~~51~~ 53 ~~55~~ ~~57~~ 59 61 ~~63~~ ~~65~~ 67 ~~69~~
71 73 ~~75~~ 77 79 ~~81~~ 83 ~~85~~ ~~87~~ 89 91 ~~93~~ ~~95~~ 97 ~~99~~

Now 7 is the next smallest uncrossed number, so we cross out starting from 7^2: 49, 63, 77, 91. Note that 63 was previously crossed out, so only three additional numbers are crossed.

3 5 7 ~~9~~ 11 13 ~~15~~ 17 19 ~~21~~ 23 ~~25~~ ~~27~~ 29 31 ~~33~~ ~~35~~
37 ~~39~~ 41 43 ~~45~~ 47 ~~49~~ ~~51~~ 53 ~~55~~ ~~57~~ 59 61 ~~63~~ ~~65~~ 67 ~~69~~
71 73 ~~75~~ ~~77~~ 79 ~~81~~ 83 ~~85~~ ~~87~~ 89 ~~91~~ ~~93~~ ~~95~~ 97 ~~99~~

Now the next smallest uncrossed number is 11, but $11^2 = 121$ is greater than 100, so sieving is complete. The list of primes is 2 followed by the remaining uncrossed numbers: 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, and 97.

The algorithm outputs primes in Step 1 (the only even prime 2), Step 3 (the sieving primes), and Step 6 (sweeping up the primes that survived the sieve). The word "output" can mean anything. It is common to collect the primes in a list, but depending on your needs, you could count them, sum them, or use them in many other ways.

The Sieve of Eratosthenes runs in time $O(n \log \log n)$, and because the only operations in the inner loop (Step 4) are a single comparison, addition and crossing-out, it is very fast in practice.

There are other ways to make lists of prime numbers. If memory is constrained, or if you want only the primes on a limited range from m to n, you may be interested in the segmented Sieve of Eratosthenes, which finds the primes in blocks; the sieving primes are those less than $\sqrt{n}$, and the minimum multiple of each sieving prime in each segment is reset at the beginning of the next segment. A. O. L. Atkin, an IBM researcher, invented a sieve that is faster than the Sieve of Eratosthenes, as it crosses out multiples of the squares of the sieving primes after some precomputations. Paul Pritchard, an Australian mathematician, has developed several methods of sieving using wheels that have time complexity $O(n/\log \log n)$, which is asymptotically faster than the Sieve of Eratosthenes, though in practice Pritchard's sieves are somewhat slower since the bookkeeping in each step of the inner loop is more involved.

Sometimes instead of the primes less than n you need the first x primes. The simplest method is to estimate the value of n using the Prime Number Theorem, first conjectured by Carl Friedrich Gauss in 1792 or 1793 (at the age of fifteen) and proved independently by Jacques Hadamard and Charles Jean de la Vallée-Poussin in 1896, which states that there are approximately $x/log_e x$ primes less than x. A corollary to the Prime Number Theorem states that, for $n > 5$, the nth prime number P_n is on the range $n \log n < P_n < n(\log n + \log \log n)$ so to compute the first x primes, sieve to $n(\log n + \log \log n)$ and discard the excess.

By the way, many people implement a function to list the prime numbers to a limit that they call the Sieve of Eratosthenes, but really isn't; their functions use trial division instead. If you use a modulo operator, or division in any form, your algorithm is *not* the Sieve of Eratosthenes, and will run much slower than the algorithm described above. On a recent-vintage personal computer, a properly-implemented Sieve of Eratosthenes should be able to list the 78498 primes less than a million in less than a second.

2 Trial Division

We turn next to the problem of classifying a number n as prime or composite. The oldest method, and for nearly two thousand years the only method, was trial division. If n has no remainder when divided by 2, it is composite. Otherwise, if n has no remainder when divided by 3, it is composite. Otherwise, if n has no remainder when divided by 4, it is composite. Otherwise, ... And so on. Iteration stops, and the number is declared prime, when the trial divisor is greater than $\sqrt{n}$. We can see that this is so because, if $n = p \cdot q$, then one of p or q must be less than $\sqrt{n}$ while the other is greater than $\sqrt{n}$ (unless p and q are equal, and n is a perfect square). A simple optimization notes that if the number is even it is composite, and if the number is odd any factor must be odd, so it is necessary to divide only by the odd numbers greater than 2, not by the even numbers.

Algorithm 2.A: Primality Testing by Trial Division: Determine if an integer $n > 1$ is prime or composite:

1. [*Finished if even*] If n is even, return `COMPOSITE` and stop.
2. [*Initialize for odd*] Set $d \leftarrow 3$.
3. [*Terminate if prime*] If $d^2 > n$, return `PRIME` and stop.
4. [*Trial division*] If $n \equiv 0 \pmod{d}$, return `COMPOSITE` and stop.
5. [*Iterate*] Set $d \leftarrow d + 2$ and go to Step 3.

As an example, consider the number 100524249167. It is not even. Dividing by 3 gives a remainder of 2. Dividing by 5 gives a remainder of 2. Dividing by 7 gives a remainder of 6. Dividing by 9 gives a remainder of 5. Dividing by 11 gives a remainder of 1. Dividing by 13 gives a remainder of 4. Dividing by 15 gives a remainder of 2. Dividing by 17 gives a remainder of 8. Dividing by 19 gives a remainder of 3. Dividing by 21 gives a remainder of 20. Dividing by 23 gives a remainder of 0 and, in Step 4, demonstrates that $100524249167 = 23 \cdot 127 \cdot 239 \cdot 311 \cdot 463$ is composite.

There are other optimizations possible. If you have a list of prime numbers at hand, you can use only the primes as trial divisors and skip the composites, which makes things go quite a bit faster; this works because, if you've already tested its component primes, it is not possible for a composite to be a divisor. If you can't afford the space to store a list of primes, you can use one of Pritchard's wheels, just as with the sieve.

Trial division iterates until it reaches either the smallest prime factor of a number or its square root. Most composites are identified fairly quickly; it's the primes that take longer, as all the trial divisors fail one by one. In general, the time complexity of trial division is $O(\sqrt{n})$, and if n is large it will

take a very long time. Thus, trial division is best limited to cases where n is small, less than a billion or thereabouts. If n is large, it is common to use probabilistic methods to distinguish primes from composites, as a later section will show.

Another problem for which trial division provides a solution is the problem of breaking down a composite integer into its prime factors. The ancient Greek mathematicians proved the Fundamental Theorem of Arithmetic that the factorization of a number is unique, ignoring the order of the factors. Modern mathematicians use the difficulty of the factorization process to provide cryptographic security for the internet. Factoring integers is a hard and honorable problem.

Euclid gave the proof of the Fundamental Theorem of Arithmetic in his book *Elements*. The proof is in four parts. The first paragraph, from Book VII, Proposition 20, proves the uniqueness of fractions, and is used in the second paragraph that proves Book VII, Proposition 30, now known as Euclid's Lemma, that if a prime p divides the product ab then either p divides a or p divides b. The third paragraph proves that all positive integers greater than 1 can be written as the product of primes, and the fourth paragraph proves that the factorization is unique, which is the Fundamental Theorem of Arithmetic, stated by Euclid in Book IX, Proposition 14.

> Suppose that a ratio $a : b$ reduces to $c : d$ in lowest terms, assume that c does not divide a, and assume that $c(m/n) = a$. Since $a : b$ is the same ratio as $c : d$, then $d(m/n) = b$, which implies that $c/m = (1/n)a$ and $d/m = (1/n)b$. Therefore $c/m : d/m$ is the same ratio as $a : b$, which shows that $c : d$ is not in lowest terms. But that contradicts the earlier assumption. Therefore c does divide a, and d divides b the same number of times.
>
> Suppose that a prime p divides the product ab but that p does not divide a, so that p is relatively prime to a. Let $n = ab/p$; this must be an integer because $p|ab$. Then $p/a = b/n$, and p/a must be in lowest terms, because p is relatively prime to a. Thus, by the previous paragraph, there must be some x for which $px = b$ and $ax = n$, because the two ratios are the same, and therefore $p|b$. Likewise, if we suppose that p does not divide b, then $p|a$. Thus, if p is prime and $p|ab$, then either $p|a$ or $p|b$.
>
> Suppose that n is the smallest positive integer greater than 1 that cannot be writen as the product of primes. Now n cannot be prime because such a number is the product of a single prime, itself. Thus the composite is $n = a \cdot b$, where both a and b are positive integers less than n. Since n is the smallest number that cannot be written as the product of primes, both a and b must be able to be written as the product of primes. But then $n = a \cdot b$ can be written as the product of primes simply by combining the factorizations of a and b. But that contradicts our supposition. Therefore, all positive integers greater than 1 can be written as the product of primes.
>
> Furthermore, that factorization is unique, ignoring the order in which the primes are written. Now suppose that s is the smallest positive integer greater than 1 that can be written as two different products of prime numbers, so that $s = p_1 \cdot p_2 \cdot \ldots \cdot p_m = q_1 \cdot q_2 \cdot \ldots \cdot q_n$. By Euclid's Lemma either p_1 divides q_1 or p_1 divides $q_2 \cdot \ldots \cdot q_n$. Therefore $p_1 = q_k$ for some k. But removing p_1 and q_k from the initial equivalence leaves a smaller integer that can be factored in two ways, contradicting the initial supposition. Thus there can be no such s, and all integers greater than 1 have a unique factorization. —Euclid, *Elements*, Book IX, Proposition 14, circa 300 B.C.

There are many algorithms for factoring integers, of which the simplest is trial division. Divide the number being factored by 2, then 3, then 4, and so on. When a factor divides evenly, record it, divide the original number by the factor, then continue the process until the remaining cofactor is prime. A simple optimization notes that once the factors of 2 have been removed from a number, it is odd, and all its factors will be odd, so it is only necessary to perform trial division by 2 and the odd numbers starting from 3, thus halving the work required to be done. A second optimization stops the trial division when the factor being tried exceeds the square root of the current cofactor, indicating that the current cofactor is prime and is thus the final factor of the original number. Here is a formal statement of the trial division factorization algorithm:

Algorithm 2.B: Integer Factorization by Trial Division: Find the prime factors of a composite integer $n > 1$:

1. [*Remove factors of 2*] If n is even, output the factor 2, set $n \leftarrow n/2$, and go to Step 1.
2. [*Initialize for odd factors*] If $n = 1$, stop. Otherwise, set $f \leftarrow 3$.
3. [*Terminate if prime*] If $n < f^2$, output the factor n and stop.
4. [*Trial division*] Calculate the quotient q and remainder r when dividing n by f, so that $n = qf + r$ with $0 \le r < f$.
5. [*Loop on odd integers*] If $r > 0$, set $f \leftarrow f + 2$ and go to Step 3. Otherwise (if $r = 0$), output the factor f, set $n \leftarrow q$, and go to Step 3.

As an example, we find the factors of $n = 13195$. Since 13195 is odd, Step 1 does nothing and we go to Step 2, where $f = 3$. Since $3^2 < 13195$, we calculate $q = 13195/3 = 4398$ and $r = 1$ in Step 4, so in Step 5 we set $f = 3+2 = 5$ and go to Step 3. Since $5^2 < 13195$, we calculate $q = 13195/5 = 2639$ and $r = 0$ in Step 4, so in Step 5 we output the factor 5, set $n = 2639$, and go to Step 3. Since $5^2 < 2639$, we calculate $q = 2639/5 = 527$ and $r = 4$ in Step 4, so in Step 5 we set $f = 5+2 = 7$ and go to Step 3. Since $7^2 < 2639$, we calculate $q = 2639/7 = 377$ and $r = 0$ in Step 4, so in Step 5 we output

the factor 7, set $n = 377$, and go to Step 3. Since $7^2 < 377$, we calculate $q = 377/7 = 53$ and $r = 6$ in Step 4, so in Step 5 we set $f = 7 + 2 = 9$ and go to Step 3. Since $9^2 < 377$, we calculate $q = 377/9 = 41$ and $r = 8$ in Step 4, so in Step 5 we set $f = 9 + 2 = 11$ and go to Step 3. Since $11^2 < 377$, we calculate $q = 377/11 = 34$ and $r = 3$ in Step 4, so in Step 5 we set $f = 11 + 2 = 13$ and go to Step 3. Since $13^2 < 377$, we calculate $q = 377/13 = 29$ and $r = 0$ in Step 4, so in Step 5 we output the factor 13, set $n = 29$, and go to Step 3. Since $29 < 13^2$, we output the factor 29 in Step 3 and stop. The complete factorization is $13195 = 5 \cdot 7 \cdot 13 \cdot 29$.

Step 1 removes factors of 2 from n. If you like, you can extend that to other primes: remove factors of 3, then factors of 5, then factors of 7, and so on, never wasting the time to trial-divide by a composite. Of course, that requires you to pre-compute and store the primes up to $\sqrt{n}$, which may be inconvenient. If you prefer, there is a method known as wheel factorization, akin to Pritchard's sieving wheels, that achieves most of the benefits of trial division by primes but requires only a small, constant amount of extra space to store the wheel.

The time complexity of trial division is $O(\sqrt{n})$, as all trial divisors up to the square root of the number being factored must potentially be tried. In practice, you will generally want to choose a bound on the maximum divisor you are willing to test, depending on the speed of your hardware and on your patience; generally speaking, the bound should be fairly small, perhaps somewhere between a thousand and a million. Then, if you have reached the bound without completing the factorization, you can turn to a different, more potent, method of integer factorization.

3 Pseudoprimality Checking

As we saw above, trial division can be used to determine if a number n is prime or composite, but is very slow. Often it is sufficient to show that n is *probably* prime, which is a very much faster calculation; the French number theorist Henri Cohen calls numbers that pass a probable-prime test *industrial-grade* primes. The most common method of testing an industrial-grade prime is based on a theorem that dates to 1640.

Pierre de Fermat was a French lawyer at the *Parlement* in Toulouse, a jurist, and an amateur mathematician who worked in number theory, probability, analytic geometry, differential calculus, and optics. Fermat's Little Theorem states that if p is a prime number, then for any integer a it is true that $a^p \equiv a \pmod{p}$, which is frequently stated in the equivalent form $a^{p-1} \equiv 1 \pmod{p}$ by dividing both sides of the congruence by a, assuming $a \neq 0$. As was his habit, Fermat gave no proof for his theorem, which was proved by Gottfried Wilhelm von Leibniz forty years later and first published by Leonhard Euler in 1736. This formula gives us a way to distinguish primes from composites: if we can find an a for which Fermat's Little Theorem fails, then p must be composite.

But there is a problem. There are some numbers, known as Carmichael numbers, that are composite but pass Fermat's test for all a; the smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$, and the sequence begins 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, ... In 1976, Gary Lee Miller, a computer science professor at Carnegie Mellon University, developed an alternate test for which there are no *strong liars*, that is, there are no numbers for which there is no a that distinguishes prime from composite. A strong pseudoprime to base a is an odd composite number $n = d2^s + 1$ with d odd for which either $a^d \equiv 1 \pmod{n}$ or $a^{d \cdot 2^r} \equiv -1 \pmod{n}$ for some $r = 0, 1, \ldots, s-1$. This works because $n = 2m + 1$ is odd, so we can rewrite Fermat's Little Theorem as $a^{2m} - 1 \equiv (a^m - 1)(am + 1) \equiv 0 \pmod{n}$. If n is prime, it must divide one of the factors, but can't divide both because it would then divide their difference $(a^m + 1) - (a^m - 1) = 2$. Miller's observation leads to the strong pseudoprime test.

Algorithm 3.A: Strong Pseudoprime Test: Determine if a on the range $1 < a < n$ is a witness to the compositeness of an odd integer $n > 2$:

1. [*Initialize*] Set $d \leftarrow n - 1$. Set $s \leftarrow 0$.
2. [*Reduce while even*] If d is even, set $d \leftarrow d/2$, set $s \leftarrow s + 1$, and go to Step 2.
3. [*Easy return?*] Set $t \leftarrow a^d \pmod{n}$. If $t = 1$ or $t = n - 1$, output `PROBABLY PRIME` and stop.
4. [*Terminate?*] Set $s \leftarrow s - 1$. If $s = 0$, output `COMPOSITE` and stop.
5. [*Square and test*] Set $t \leftarrow t^2 \pmod{n}$. If $t = n - 1$, output `PROBABLY PRIME` and stop. Otherwise, go to Step 4.

The algorithm is stated differently than the math given above, though the result is the same. In the math, we calculated $a^{d \cdot 2^r}$, which is initially a^d when $r = 0$, then a^{2d}, then a^{4d}, and so on; in other words, a^d is squared at each step. Step 5 thus reduces the modular operation from exponentiation to multiplication; the strength reduction makes the code simpler and faster.

As an example, consider the prime number $73 = 23 \cdot 9 + 1$; at the end of Step 2, $d = 9$ and $s = 3$. If the witness is 2, then $29 \equiv 1 \pmod{73}$ and 73 is declared `PROBABLY PRIME` in Step 3. If the witness is 3, then $3^9 \equiv 46 \pmod{73}$ and the test of Step 3 is indeterminate, but $3^{2 \cdot 9} \equiv 72 \equiv -1 \pmod{73}$ in the second iteration of Step 5, and 73 is declared `PROBABLY PRIME`. On the other hand, the composite number $75 = 2^1 \cdot 37 + 1$ is declared `COMPOSITE` with witness 2 because$2^{37} \equiv 47 \pmod{75}$ in Step 3. One more example is the composite number $2047 = 23 \cdot 89$, which is declared `PROBABLY PRIME` by the witness 2 but `COMPOSITE` by the witness 3; 2047 is the smallest number for which 2 is a strong liar to its compositeness.

Miller proved that n must be prime if no a from 2 to $70(\log_e n)^2$ is a witness to the compositeness of n; Eric Bach,

a professor at the University of Wisconsin in Madison, later reduced the constant from 70 to 2. Unfortunately, the proof assumes the Riemann Hypothesis and can't be relied upon because the Riemann Hypothesis remains unproven. However, Michael O. Rabin, an Israeli computer scientist and recipient of the Turing Award, used Miller's strong pseudoprime test to build a probabilistic primality test. Rabin proved that for any odd composite n, at least 3/4 of the bases a are witnesses to the compositeness of n; although that's the proven lower bound, in practice the proportion is much higher than 3/4. Thus, the Miller-Rabin method performs k strong pseudoprime tests, each with a different a, and if all the tests pass the method concludes that n is prime with probability at least 4^{-k}, and in practice much higher; a common value of k is 25, which gives a maximum probability of 1 error in 10^{17}.

Algorithm 3.B: Miller-Rabin Pseudoprimality Test: Determine if an odd integer n is prime with a probability of at least 4^{-k} by performing k strong pseudoprime tests:

1. [*Terminate?*] If $k = 0$, output `PROBABLY PRIME` and stop.

2. [*Strong-pseudoprime test*] Choose a random number a such that $1 < a < n$. Perform a strong pseudoprime test using Algorithm 3.A to determine if a is a witness to the compositeness of n.

3. [*Pseudoprime?*] If the strong pseudoprime test indicates a is a witness to the compositeness of n, output `COMPOSITE` and stop. Otherwise, set $k \leftarrow k - 1$ and go to Step 1.

Although the algorithm given above specifies random numbers for the bases of the strong pseudoprime test, it is common to fix the bases in advance, based on the value of n. If n is a 32-bit integer, it is sufficient to test on the three bases 2, 7, and 61; all the odd numbers less than 2^32 have been tested and no errors in the determination of primality exist. If n is less than a trillion, it is sufficient to test to the bases 2, 13, 23, and 1662803. Gerhard Jaesche used the first seven primes as bases and determined that the first false positive is 341550071728321. And Zhenxiang Zhang plausibly conjectures that there are no errors less than 10^36 when using the first twenty primes as bases.

As an example, we determine the primality of $2^{149} - 1$. By Algorithm 3.A, 3 is a witness that $2^{149} - 1 = 86652685662821831510 \cdot 8235109336690846723986161$ is composite. That determination would be impossible for trial division, at least in any reasonable time frame, as the Prime Number Theorem suggests there are approximately $1.9 \cdot 10^{18}$ primes to be tested.

Steps 3 and 5 of Algorithm 3.A require modular exponentiation. Some languages provide modular exponentiation as a built-in function, but others don't. If your language doesn't, you will have to write your own function. You should not write your function by first performing the exponentiation and then performing the modulo operation, as the intermediate result of the exponentiation can be very large. Instead, use the square-and-multiply algorithm.

Algorithm 3.C: Modular Exponentiation: Compute $b^e \pmod m$ with b, e and m all positive integers:

1. [*Initialize*] Set $r \leftarrow 1$.

2. [*Terminate?*] If $e = 0$, output r and stop.

3. [*Multiply if odd*] If e is odd, set $r \leftarrow r \cdot b \pmod m$.

4. [*Square and iterate*] Set $e \leftarrow \lfloor e/2 \rfloor$. Set $b \leftarrow b^2 \pmod m$. Go to Step 2.

Consider the calculation $437^{13} \pmod{1741} = 819$. Initially $b = 437$, $e = 13$, $r = 1$ and the test in Step 2 fails. Since e is odd, $r = 1 \cdot 437 = 437$ in Step 3, then $e = 13/2 = 6$ and $b = 437^2 \pmod{1741} = 1200$ in Step 4 and the test in Step 2 fails. Since e is even, $r = 437$ is unchanged in Step 3, then $e = 6/2 = 3$ and $b = 1200^2 \pmod{1741} = 193$ in Step 4 and the test in Step 2 fails. Since e is odd, $r = 437 \cdot 193 \pmod{1741} = 773$ in Step 3, then $e = 3/2 = 1$ and $b = 193^2 \pmod{1741} = 688$ in Step 4 and the test in Step 2 fails. Since e is odd, $r = 773 \cdot 688 \pmod{1741} = 819$ in Step 3, then $e = 1/2 = 0$ and $b = 688^2 \pmod{1741} = 1533$ in Step 4. At this point the test in Step 2 succeeds and the result $r = 819$ is returned. By the way, the intermediate calculation results in the very large number $437^{13} = 211962327928904762351644463150065 97$, so you can see why Algorithm 3.C is preferable.

The time complexity of the Miller-Rabin primality checker is $O(1)$, which is vastly better than trial division. The time for the strong pseudoprime test depends on the number of factors of 2 found in $n - 1$, which is independent of n. Likewise, the number k of strong pseudoprime tests is independent of n, so it contributes to the implied constant, not to the overall order. If n is large, the arithmetic takes time $O(\log \log n)$, but we ignore that in our analysis.

There are other methods for quickly checking the primality of a number, including the Baillie-Wagstaff method that combines a strong pseudoprime test base 2 with a Lucas pseudoprime test and the method of Mathematica that adds a strong pseudoprime test base 3 to the Baillie-Wagstaff method; both methods are faster than the Miller-Rabin method, and also give fewer false positives. If a slight chance of error is too much for you, and you need to *prove* the primality of a number, you can use the trial division of Algorithm 2.A, there is a method of Pocklington that uses the factorization of $n-1$, a method using Jacobi sums (the APR-CL method), a method using elliptic curves due to Atkin and Morain, and the new AKS method, which operates in proven polynomial time but is not yet practical.

4 Pollard's Rho Method

In 1975, British mathematician John Pollard invented a method of integer factorization that finds factors in time

$O(\sqrt[4]{n})$, which is the square root of the $O(\sqrt{n})$ time complexity of trial division. The method is simple to program and takes only a small amount of auxiliary space. Before we explain Pollard's algorithm, we discuss two elements of mathematics on which it relies, the Chinese Remainder Theorem and the birthday paradox.

The original version of the Chinese Remainder Theorem was stated in the third-century by the Chinese mathematician Sun Zi (his name is sometimes spelled Sun Tsu or Sun Tzu, but he is not the same person as the famous writer on the art of war) in his book *Sun Zi suanjing* (literally, *The Mathematical Classic of Sun Zi*), and was proved by the Indian mathematician Aryabhata in the sixth century:

> Let r and s be positive integers which are relatively prime and let a and b be any two integers. Then there exists an integer n such that $n \equiv a \pmod{r}$ and $n \equiv b \pmod{s}$. Furthermore, n is unique modulo the product $r \cdot s$.

Sun Zi gave an example: When a number is divided by 3, the remainder is 2. When the same number is divided by 5 the remainder is 3. And when the same number is divided by 7, the remainder is 2. The smallest number that satisfies all three criteria is 23, which you can verify easily. And since the least common multiple of 3, 5, and 7 is 105, any number of the form $23 + 105k$, from the arithmetic progression 23, 128, 233, 338, ..., is also a solution. Sun Zi used this method to count the men in his emperor's armies; arrange them in columns of 11, then 12, then 13, take the remainder at each step, and calculate the number of soldiers.

In the theory of probability, the "birthday paradox" calculates the likelihood that in a group of p people two of them will have the same birthday. Obviously, in a group of 367 people the probability is 100%, since there are only 366 possible birthdays. What is surprising is that there is a 99% probability of a matching pair in a group as small as 57 people and a 50% probability of a matching pair in a group as small as 23 people. If, instead of birthdays, we consider integers modulo n, there is a 50% probability that two integers are congruent modulo n in a group of $1.177\sqrt{n}$ integers.

Pollard's rho algorithm uses the quadratic congruential random-number generator $x^2 + c \pmod{n}$ with $c \notin \{0, -2\}$ to generate a series of random integers x_k. By the Chinese Remainder Theorem, if $n = p \cdot q$, then $x \pmod{n}$ corresponds uniquely to the pair of integers $x \pmod{p}$ and $x \pmod{q}$. Furthermore, the x_k sequence also follows the Chinese Remainder Theorem, so that $x_{k+1} = [x_k \pmod{p}]^2 + c \pmod{p}$ and $x_{k+1} = [x_k \pmod{q}]^2 + c \pmod{q}$, so that the sequence of x_k falls into a much shorter cycle of length $\sqrt{p}$ by the birthday paradox. Thus p is identified when x_k and x_{k+1} are congruent modulo p, which can be determined when $\gcd(|x_k - x_{k+1}|, n) = p$ is between 1 and n.

Depending on the values of p, q and c, it is possible that the random-number generator may reach a cycle before a factor is found. Thus, Pollard used Robert Floyd's tortoise-and-hare cycle-detection method. The sequence of x_k starts with two values the same, call them t and h. Then each time t is incremented, h is incremented twice; the hare runs twice as fast as the tortoise. If the hare reaches the tortoise, that is, $t \equiv h \pmod{n}$, before a factor is found, then a cycle has been reached and further work is pointless. At that point, either the factorization attempt can be abandoned or a new random-number generator can be tried using a different c.

Pollard called his method "Monte Carlo factorization" because of the use of random numbers. The algorithm is now called the rho algorithm because the sequence of x_k values has an initial tail followed by a cycle, giving it the shape of the Greek letter rho ρ. Fortunately the algorithm is much simpler than the explanation.

Algorithm 4.A: Pollard's Rho Method: Find a factor of an odd composite integer $n > 1$:

1. [*Initialization*] Set $t \leftarrow 2$, $h \leftarrow 2$ and $c \leftarrow 1$. Define the function $f(x) = x^2 + c \pmod{n}$.
2. [*Iteration*] Set $t \leftarrow f(t)$, $h \leftarrow f(f(h))$, and $d \leftarrow \gcd(t - h, n)$. If $d = 1$, go to Step 2.
3. [*Termination*] If $d < n$, output d and stop. Otherwise, either stop with failure or continue by setting $t \leftarrow 2$, $h \leftarrow 2$, and $c \leftarrow c + 1$, redefining the function $f(x)$ using the new value of c and going to Step 2.

As an example, we consider the factorization of 8051. Initially, $t = 2$, $h = 2$, and $c = 1$. After one iteration of Step 2, $t = 2^2 + 1 = 5 \pmod{8051}$, $h = (2^2 + 1)^2 + 1 = 26 \pmod{8051}$, and $d = \gcd(5 - 26, 8051) = 1$. After the second iteration of Step 2, $t = 5^2 + 1 = 26 \pmod{8051}$, $h = (26^2 + 1)^2 + 1 = 7474 \pmod{8051}$, and $d = \gcd(26 - 7474, 8051) = 1$. After the third iteration of Step 2, $t = 26^2 + 1 = 677 \pmod{8051}$, $h = (7474^2 + 1)^2 + 1 = 871 \pmod{8051}$, and $d = gcd(677 - 871, 8051) = 97$, which is a factor of 8051; the complete factorization is $8051 = 83 \cdot 97$.

Be sure before you begin that n is composite; if n is prime, then d will always be 1 (because if n is prime it is always coprime to every number smaller than itself) and the algorithm will loop forever. As with trial division, it is probably wise to set some bound on the maximum number of steps you are willing to take in the iteration of Step 2, because large factors can take a long time to find using this algorithm. You should also be careful not to let c be 0 or -2, because in those cases the random numbers aren't very random. Note that the factor found in Step 3 may not be prime, in which case you can apply the algorithm again to the reduced factor, using a different c. And of course, once you have one factor, you can continue by factoring the remaining cofactor.

Algorithm 4.B: Integer Factorization by Pollard's Rho Method: Find all the prime factors of a composite integer $n > 1$:

1. [*Remove factors of 2*] If n is even, output the factor 2, set $n \leftarrow n/2$, and go to Step 1.
2. [*Terminate if prime*] If n is prime by the method of Algorithm 3.B, output the factor n and stop.

3. [*Find a factor*] Use Algorithm 4.A to find a factor of n and call it f. Output the factor f, set $n \leftarrow n/f$, and go to Step 2.

There are two ways in which Pollard's algorithm can be improved. First, it should bother you that each number in the random sequence is computed twice; it bothered the Australian mathematician Richard Brent, who devised a cycle-finding algorithm based on powers of 2 that computes each number in the random sequence only once, and it is Brent's variant that is most often used today. A second improvement notes that for any a, b, and n, $\gcd(ab, n) > 1$ if and only if at least one of $\gcd(a, n) > 1$ or $\gcd(b, n) > 1$, and accumulates the products of the elements of the t and h sequences for several steps (for large n, 100 steps is common) before computing the gcd, thus saving much time; if the gcd is n, then it is possible either that a cycle has been found or that two factors were found since the last gcd, in which case it is necessary to return to values saved from the previous gcd calculation and iterate one step at a time.

The time complexity of Pollard's rho algorithm depends on the unknown factor d. By the birthday paradox, in the average case it will take $1.177\sqrt{d}$ steps to find the factor, or $O(\sqrt{d})$. Thus, if n is the product of two primes, it will take $O(\sqrt[4]{n})$ to perform the factorization, assuming the two primes are roughly the same size. In other words, a million iterations of trial division will find factors up to a million, while a million iterations of Pollard's rho method will find factors up to a trillion; that's why you want to switch from trial division to Pollard's rho method at a fairly low bound.

5 Going Further

Although there is more to programming with prime numbers, we will stop here, since our small library has fulfilled our modest goals. The five appendices give implementations in C, Haskell, Java, Python, and Scheme, and the savvy reader will study all of them, because while they all implement exactly the same algorithms, each does so in a different way, and the differences are enlightening, about both the algorithms and the languages. The C appendix describes the tasteful use of the GMP multi-precision number library. The Haskell and Scheme appendices describe some of the syntax and semantics of those languages, on the assumption that they are unfamiliar to many readers. The Java appendix is the most faithful of all the appendices to the exact structure of the algorithms, including error-checking on the inputs as described in the preambles of each of the algorithms. The Python and Scheme appendices are the most "real-world" implementations, as they include error-checking on the inputs, bounds-checking to stop calculations that take too long, and even a non-mathematical but highly useful extension of the domain of the factoring functions.

Although our goals were modest, we have accomplished much. It's hard to improve on the optimized Sieve of Eratosthenes, and the Miller-Rabin primality checker will handle inputs of virtually unlimited size. The rho algorithm will find most factors up to a dozen digits or more, regardless of the size of the number being factored.

If Pollard's rho algorithm won't crack your composite, there are more powerful algorithms available, though they are beyond our modest aspirations. The elliptic curve method will find factors up to about thirty or forty digits (even fifty or sixty digits if you are patient). The quadratic sieve will split semi-primes up to about 90 digits on a single personal computer or 120 digits on a modest network of personal computers, and the number field sieve will split semi-primes up to about 200 digits on that same network. At the time this of this writing, the current record factorization is 231 decimal digits (768 bits), which took a team of experts about 2000 PC-years, and about eight months of calendar time, on a "network" of computers around the world connected via email.

If your goal isn't self-study and you really want to factor a large number, and the rho technique fails, you have several options. A good first step is Dario Alpern's factorization applet at http://www.alpertron.com.ar/ECM.HTM. Paul Zimmermann's gmp-ecm program at http://ecm.gforge.inria.fr uses a combination of trial division, Pollard's rho algorithm, another algorithm of Pollard known as $p - 1$, and Hendrik Lenstra's elliptic curve method to find factors. Jason Papadopoulos' msieve program at http://www.boo.net/~jasonp/qs.html uses both the quadratic sieve of Carl Pomerance and the number field sieve of John Pollard.

There is much more to prime numbers and integer factorization than we have discussed here; for instance, there are methods other than trial division for proving the primality of large numbers (several hundred digits) and methods other than enumeration with a sieve for counting the primes less than a given input number. At the end of each algorithm above was a discussion of alternatives; the interested reader will find that web searches for the topics mentioned will be fruitful and interesting. A superb reference for programmers is the book *Prime Numbers: A Computational Perspective* by Richard Crandall and Carl B. Pomerance (be sure to look for the second edition, which includes discussion of the new AKS primality prover); beware that although the approach is computational, there is still heavy mathematical content in the book. You may also be interested in the Programming Praxis web site at http://programmingpraxis.com, which has many exercises on the theme of prime numbers.

Appendix: C

C is a small language with limited data types; integers are limited to what the underlying hardware provides, there are no lists, and there are no bitarrays, so we have to depend on libraries to provide those things for us. Since we are interested in prime numbers and not lists or bitarrays, we will write the smallest libraries that are necessary to get a working program. We begin with bitarrays, which are represented as arrays of characters in which we set and clear individual bits using macros:

```
#define ISBITSET(x, i) (( x[i>>3] & (1<<(i&7)) ) != 0)
#define SETBIT(x, i) x[i>>3] |= (1<<(i&7));
#define CLEARBIT(x, i) x[i>>3] &= (1<<(i&7)) ^ 0xFF;
```

To declare a bitarray b of length $8n$, say `char b[n]`, and to initialize each bit to 0, say `memset(b, 0, sizeof(b))`; change the 0 to 255 to set each bit initially to 1. Here is our minimal list library:

```
typedef struct list {
    void *data;
    struct list *next;
} List;

List *insert(void *data, List *next)
{
    List *new;

    new = malloc(sizeof(List));
    new->data = data;
    new->next = next;
    return new;
}

List *insert_in_order(void *x, List *xs)
{
    if (xs == NULL || mpz_cmp(x, xs->data) < 0)
    {
        return insert(x, xs);
    }
    else
    {
        List *head = xs;
        while (xs->next != NULL &&
               mpz_cmp(x, xs->next->data) > 0)
        {
            xs = xs->next;
        }
        xs->next = insert(x, xs->next);
        return head;
    }
}

List *reverse(List *list) {
    List *new = NULL;
    List *next;

    while (list != NULL)
    {
        next = list->next;
        list->next = new;
        new = list;
        list = next;
    }

    return new;
}

int length(List *xs)
{
    int len = 0;
    while (xs != NULL)
    {
        len += 1;
        xs = xs->next;
    }
    return len;
}
```

Lists are represented as structs of two members; the empty list is represented as `NULL`. The trickiest function is `reverse`, which operates in-place to make each list item point to its predecessor. Function `insert_in_order` assumes numbers represented using the Gnu multi-precision library, which we will see shortly. The list functions leave all notion of memory management to the caller.

With that out of the way, we are ready to begin work on the prime number functions. The ancient Sieve of Eratosthenes uses b for the bitarray. The outer `for` loop indexes through the bitarray with p and the inner `for` loop sieves the multiples of p at index locations i. The final `reverse` is necessary because primes are added to the outgoing list working back to front.

```
List *sieve(long n)
{
    char b[(n+1)/8+1];
    int i, p;
    List *ps = NULL;

    memset(b, 255, sizeof(b));

    for (p=2; p<=n; p++)
    {
        if (ISBITSET(b,p))
        {
            ps = insert((void *) p, ps);
            for (i=p; i<=n; i+=p)
            {
                CLEARBIT(b,i);
            }
        }
    }

    return reverse(ps);
}
```

The optimized Sieve of Eratosthenes uses b for the bitarray, i indexes into the bitarray, and $p = 2i + 3$ is the number represented at location i of the bitarray. The first `while` loop identifies the sieving primes and performs the sieving in an inner `while`, and the third `while` loop sweeps up the remaining primes that survive the sieve.

```
List *primes(long n)
{
    int m = (n-1) / 2;
    char b[m/8+1];
    int i = 0;
    int p = 3;
    List *ps = NULL;
    int j;

    ps = insert((void *) 2, ps);

    memset(b, 255, sizeof(b));

    while (p*p < n)
    {
        if (ISBITSET(b,i))
        {
            ps = insert((void *) p, ps);
            j = (p*p - 3) / 2;
            while (j < m)
            {
                CLEARBIT(b, j);
                j += p;
            }
        }
        i += 1; p += 2;
    }

    while (i < m)
    {
        if (ISBITSET(b,i))
        {
            ps = insert((void *) p, ps);
        }
        i += 1; p += 2;
    }

    return reverse(ps);
}
```

We look next at the two algorithms that use trial division; we'll look at them together because they are so similar. We used long integers for the Sieve of Eratosthenes because they are almost certainly big enough, but we will use `long long`

`unsigned` integers for the two trial division functions because that extends the range of the inputs that we can consider. The function that tests primality using trial division uses an `if` to identify even numbers, then a `while` ranges over the odd numbers d from 3 to $\sqrt{n}$.

```
int td_prime(long long unsigned n)
{
    if (n % 2 == 0)
    {
        return n == 2;
    }

    long long unsigned d = 3;

    while (d*d <= n)
    {
        if (n % d == 0)
        {
            return 0;
        }
        d += 2;
    }

    return 1;
}
```

The `td_factors` function is very similar to the `td_prime` function. The initial `if` becomes a `while`, because we no longer want to quit as soon as we find a single factor, and the body of the second `while` also changes so that it collects all the factors instead of quitting as soon as it finds a single factor; the factors are stacked in increasing order as they are discovered, hence the reversal. The list of factors, which will contain only the original input n if it is prime, is returned as the value of the function.

```
List *td_factors(long long unsigned n)
{
    List *fs = NULL;

    while (n % 2 == 0)
    {
        fs = insert((void *) 2, fs);
        n /= 2;
    }

    if (n == 1)
    {
        return reverse(fs);
    }

    long long unsigned f = 3;

    while (f*f <= n)
    {
        if (n % f == 0)
        {
            fs = insert((void *) f, fs);
            n /= f;
        }
        else
        {
            f += 2;
        }
    }

    fs = insert((void *) n, fs);
    return reverse(fs);
}
```

We used `long` integers for the Sieve of Eratosthenes and `long long unsigned` integers for the two trial-division algorithms. Those native integer types are sufficient for those function; usually only small n are required for the Sieve of Eratosthenes, and trial division is just too slow for large n. But many applications require much larger numbers, so we need a big-integer library, and we choose the GMP library from GNU, which is well-known for its useful interface and fast, bug-free implementation. You can obtain GMP from gmplib.org; to use it in your program, include the line `#include <gmp.h>` at the top of your program, and link with the option `-lgmp`.

We look next at Gary Miller's strong pseudoprime test. The first `while` computes d and s, then the `if` checks for an early return, the second `while` computes and tests the powers of a, and the default return is composite.

```
int is_spsp(mpz_t n, mpz_t a)
{
    mpz_t d, n1, t;
    mpz_inits(d, n1, t, NULL);
    mpz_sub_ui(n1, n, 1);
    mpz_set(d, n1);
    int s = 0;

    while (mpz_even_p(d))
    {
        mpz_divexact_ui(d, d, 2);
        s += 1;
    }

    mpz_powm(t, a, d, n);
    if (mpz_cmp_ui(t, 1) == 0 ||
        mpz_cmp(t, n1) == 0)
    {
        mpz_clears(d, n1, t, NULL);
        return 1;
    }

    while (--s > 0)
    {
        mpz_mul(t, t, t);
        mpz_mod(t, t, n);
        if (mpz_cmp(t, n1) == 0)
        {
            mpz_clears(d, n1, t, NULL);
            return 1;
        }
    }

    mpz_clears(d, n1, t, NULL);
    return 0;
}
```

Let's take a moment for a quick lesson in GMP. The datatype of big integers is given by `mpz_t`, where the `mp` is for multi-precision, `z` is for integer (from the German word Zahlen, for number), and `_t` is to indicate a type variable. All `mpz_t` variables must be initialized and cleared; in exchange for this effort, GMP takes care of all memory management automatically. The basic operations are given as `mpz_add`, `mpz_mul`, `mpz_powm` and the like, and they all return `void`, with the result given in the first argument (by analogy to an assignment, which puts the result on the left); the various division operators have their own naming conventions. Most of the operators have `_ui` variants in which the second operand (third argument) is a `long unsigned` integer instead of an `mpz_t` integer. Comparisons take two values and return a negative integer if the first is less than the second, a positive integer is the first is greater than the second, and 0 if the two are equal. We use `mpz_powm` instead of writing our own.

To determine whether a given integer is prime or composite we use 25 random integers, saving the GMP random state in a static variable that persists from one call of the function to the next. The algorithm expects an integer greater than

2, so that is our first test. Then we check that n is odd, and additionally that it is not divisible by 3, 5 or 7; those tests aren't strictly part of the algorithm, but they eliminate about three-quarters of all positive integers, and if they determine the compositeness of n, they are much cheaper than the full Miller-Rabin test. And if we still don't have an answer, we proceed with the full algorithm with k counting down to 0.

```
int is_prime(mpz_t n)
{
    static gmp_randstate_t gmpRandState;
    static int is_seeded = 0;

    if (! is_seeded)
    {
        gmp_randinit_default(gmpRandState);
        gmp_randseed_ui(gmpRandState, time(NULL));
        is_seeded = 1;
    }

    mpz_t a, n3, t;
    mpz_inits(a, n3, t, NULL);
    mpz_sub_ui(n3, n, 3);
    int i;
    int k = 25;
    long unsigned ps[] = { 2, 3, 5, 7 };

    if (mpz_cmp_ui(n, 2) < 0)
    {
        mpz_clears(a, n3, t, NULL);
        return 0;
    }

    for (i = 0; i < sizeof(ps) /
                sizeof(long unsigned); i++)
    {
        mpz_mod_ui(t, n, ps[i]);
        if (mpz_cmp_ui(t, 0) == 0)
        {
            mpz_clears(a, n3, t, NULL);
            return mpz_cmp_ui(n, ps[i]) == 0;
        }
    }

    while (k > 0)
    {
        mpz_urandomm(a, gmpRandState, n3);
        mpz_add_ui(a, a, 2);
        if (! is_spsp(n, a))
        {
            mpz_clears(a, n3, t, NULL);
            return 0;
        }
        k -= 1;
    }

    mpz_clears(a, n3, t, NULL);
    return 1;
}
```

The default GMP random number generator is the Mersenne Twister, which has good randomness properties and a very long period; we initialize the internal state of the random number generator with the current time (seconds since the epoch). The function `mpz_urandomm` returns in its first argument a uniformly-distributed pseudø-random non-negative integer less than its third argument, using and resetting the internal state of the random number generator in its second argument. GMP provides our `is_prime` function under the name `mpz_probab_prime_p`, but we give our own implementation anyway, so you can see how it is done.

There are two functions that implement integer factorization by pollard rho: `rho_factor` finds a single factor, and `rho_factors` performs the complete factorization. The `rho_factor` function assumes that n is odd and composite; t is the tortoise, h is the hare, d is the greatest common divisor, and r is a temporary working variable holding the difference between t and h. The function keeps cycling until it finds a prime factor, calling itself recursively with the next greater c if it reaches a cycle or finds a composite factor.

```
void rho_factor(mpz_t f, mpz_t n, long long unsigned c)
{
    mpz_t t, h, d, r;

    mpz_init_set_ui(t, 2);
    mpz_init_set_ui(h, 2);
    mpz_init_set_ui(d, 1);
    mpz_init_set_ui(r, 0);

    while (mpz_cmp_si(d, 1) == 0)
    {
        mpz_mul(t, t, t);
        mpz_add_ui(t, t, c);
        mpz_mod(t, t, n);

        mpz_mul(h, h, h);
        mpz_add_ui(h, h, c);
        mpz_mod(h, h, n);

        mpz_mul(h, h, h);
        mpz_add_ui(h, h, c);
        mpz_mod(h, h, n);

        mpz_sub(r, t, h);
        mpz_gcd(d, r, n);
    }

    if (mpz_cmp(d, n) == 0) /* cycle */
    {
        rho_factor(f, n, c+1);
    }
    else if (mpz_probab_prime_p(d, 25)) /* success */
    {
        mpz_set(f, d);
    }
    else /* found composite factor */
    {
        rho_factor(f, d, c+1);
    }

    mpz_clears(t, h, d, r, NULL);
}
```

The `rho_factors` function extracts factors of 2 in the first `while`, then calls `rho_factor` repeatedly in the second `while` until the remaining cofactor is prime. Function `rho_factors` returns the list of factors in its first argument, like all the GMP functions.

```
void rho_factors(List **fs, mpz_t n)
{

    while (mpz_even_p(n))
    {
        mpz_t *f = malloc(sizeof(*f));
        mpz_init_set_ui(*f, 2);
        *fs = insert(*f, *fs);
        mpz_divexact_ui(n, n, 2);
    }

    if (mpz_cmp_ui(n, 1) == 0) return;

    while (! (mpz_probab_prime_p(n, 25)))
    {
        mpz_t *f = malloc(sizeof(*f));
        mpz_init_set_ui(*f, 0);

        rho_factor(*f, n, 1);
        *fs = insert_in_order(*f, *fs);
        mpz_divexact(n, n, *f);
    }

    *fs = insert_in_order(n, *fs);
}
```

We will need four headers, including `gmp.h`:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <gmp.h>
```

We demonstrate the functions with this `main` function:

```
int main(int argc, char *argv[])
{
    mpz_t n;
    mpz_init(n);
    List *ps = NULL;
    List *fs = NULL;

    ps = sieve(100);
    while (ps != NULL)
    {
        printf("%d%s", (int) ps->data,
            (ps->next == NULL) ? "\n" : " ");
        ps = ps->next;
    }

    ps = primes(100);
    while (ps != NULL)
    {
        printf("%ld%s", (long) ps->data,
            (ps->next == NULL) ? "\n" : " ");
        ps = ps->next;
    }

    printf("%d\n", length(primes(1000000)));

    printf("%d\n", td_prime(600851475143LL));

    fs = td_factors(600851475143LL);
    while (fs != NULL)
    {
        printf("%llu%s",
            (unsigned long long int) fs->data,
            (fs->next == NULL) ? "\n" : " ");
        fs = fs->next;
    }

    mpz_t a;
    mpz_init(a);
    mpz_set_str(n, "2047", 10);
    mpz_set_str(a, "2", 10);
    printf("%d\n", is_spsp(n, a));

    mpz_set_str(n, "600851475143", 10);
    printf("%d\n", is_prime(n));

    mpz_set_str(n, "2305843009213693951", 10);
    printf("%d\n", is_prime(n));

    mpz_set_str(n, "600851475143", 10);
    rho_factors(&fs, n);
    while (fs != NULL) {
        printf("%s%s",
            mpz_get_str(NULL, 10, fs->data),
            (fs->next == NULL) ? "\n" : " ");
        fs = fs->next;
    }
}
```

To compile the program, say `gcc prime.c -lgmp -o prime`. If you get any warnings about the cast to `void *` you can safely ignore them, as it is always permissible to cast to `void`. Here is the output from the program, with the first two lines folded to accommodate the narrow columns:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
  67 71 73 79 83 89 97
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
  67 71 73 79 83 89 97
78498
0
71 839 1471 6857
1
0
1
71 839 1471 6857
```

Appendix: Haskell

Haskell is the classic purely functional language, far different from C. We begin our look at Haskell by examining a function that is frequently cited as the Sieve of Eratosthenes. Many texts include a definition something like this:

```
primes = sieve [2..]

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

But this is not the Sieve of Eratosthenes because it is based on division (the `mod` operator) rather than addition. It will quickly become slow as the primes grow larger; try, for instance, to extract a list of the primes less than a hundred thousand. Unfortunately, a proper implementation of the Sieve of Eratosthenes is a little bit ugly, since Haskell and arrays don't easily mix.

Most Haskell programs begin by importing various functions from Haskell's Standard Libraries, and ours is no exception; all imports must appear before any executable code. `ST` is the Haskell state-transformer monad, which provides mutable data structures, including `Control.Monad.ST` and `Data.Array.ST`. `Control.Monad` provides imperative-style `for` and `when`, and `Data.Array.Unboxed` provides arrays that store data directly, rather than with a pointer, as long as the data is a suitable type; we will be using arrays of booleans, which are suitable. Finally, `Data.List` provides a sort function for use by Pollard's rho algorithm.

```
import Control.Monad (forM_, when)
import Control.Monad.ST
import Data.Array.ST
import Data.Array.Unboxed
import Data.List (sort)
```

The Sieve of Eratosthenes is implemented using exactly the same algorithm as all the other languages, though it looks somewhat foreign to imperative-trained eyes. Functions in Haskell optionally begin with a declaration of the type of the function, and we will include one in each of our functions. Thus, `ancientSieve :: Int -> UArray Int Bool` declares an object named `ancientSieve` that has type (the double colon) that is a function (the arrow) that takes a value of type `Int` and returns a value of type `UArray Int Bool`. `Int` is some fixed-size integer based on the native machine type. `UArray` is an unboxed array; `Int` is the type of its indices and `Bool` is the type of its values. Note that typenames always begin with a capital letter, as opposed to simple variables or function names that begin with a lower-case letter.

```
ancientSieve :: Int -> UArray Int Bool
ancientSieve n = runSTUArray $ do
    bits <- newArray (2, n) True
    forM_ [2 .. n] $ \p -> do
        isPrime <- readArray bits p
        when isPrime $ do
            forM_ [2*p, 3*p .. n] $ \i -> do
                writeArray bits i False
    return bits
```

The body of the ancientSieve function is fairly atypical of Haskell code, due to its use of arrays. The first line, `runSTUArray $ do` (that parses as `run`, `ST` for the state-transformer monad, `UArray` for the unboxed array), sets up

the array processing; the array is initialized with indices from 2 to n, with all values `True`, and assigned to variable *bits*. An expression like `{forM_ [0..x] $ \i do` would be rendered in C as `for (i=0; i<=x; i++)`; the expression `[0 .. x]` expands to 0, 1, ..., x, and is evaluated lazily, as if by a list generator, so the whole list is never reified all at once. Functions `readArray` and `writeArray` fetch and store elements of an array. Variable `isPrime` is assigned either `True` or `False`, depending on the value of the element of the *bits* array with value i. In the inner loop, iteration starts at `2*p`, and each iteration steps by `p`, which is the difference between the first and second elements of the list, continuing until i is greater than n.

```
ancientPrimes :: Int -> [Int]
ancientPrimes n = [p | (p, True) <-
                  assocs $ ancientSieve n]
```

The `ancientPrimes` function is simple; `assocs` collects the elements of the `bits` in order, paired with their index, and those that are `True` are included in the out-going list of primes. The type signature indicates that the function takes an `Int` and returns a list of `Int` values, as indicated by the square brackets. The overall structure of the function is a list which has 2 as its head joined by a colon `:` to a list comprehension between square brackets `[ ...]`. The list comprehension has two parts. The expression p before the vertical bar `|` defines the elements of the output list. The generator after the bar assigns to the tuple all of the elements returned by the `assocs` function and keeps only those where the second element of the tuple is `True`, binding the first element of the tuple to the variable p that is used in the result expression.

Optimized `sieve` and `primes` functions are shown below. The only differences are in the array indexing, where the array index i is mapped to the prime number p by the expression $p = 2i + 3$.

```
sieve :: Int -> UArray Int Bool
sieve n = runSTUArray $ do
    let m = (n-1) `div` 2
        r = floor . sqrt $ fromIntegral n
    bits <- newArray (0, m-1) True
    forM_ [0 .. r `div` 2 - 1] $ \i -> do
        isPrime <- readArray bits i
        when isPrime $ do
            let a = 2*i*i + 6*i + 3
                b = 2*i*i + 8*i + 6
            forM_ [a, b .. (m-1)] $ \j -> do
                writeArray bits j False
    return bits

primes :: Int -> [Int]
primes n = 2 : [2*i+3 | (i, True) <- assocs $ sieve n]
```

It is simple to test primality by trial division because Haskell offers a simple way of generating the list of 2 followed by odd numbers. The colon operator, pronounced "cons," is the list constructor. The expression `[3,5..]` is a list constructor (anything surrounded by square brackets is a list) with 3 as its first element, 5 as its second element, and so on in an arithmetic progression that increases by $5 - 3 = 2$ at each step. The `..` operator at the end of the list expression signifies that the list goes on forever; if there is a value after the `..` operator, that is the ending value included in list.

```
tdPrime :: Int -> Bool
tdPrime n = prime (2:[3,5..])
    where prime (d:ds)
            | n < d * d      = True
            | n `mod` d == 0 = False
            | otherwise      = prime ds
```

Factorization by trial division is likewise expressed more simply in Haskell than in other languages because of the list generator syntax. The guard expressions of the local `facts` function makes `tdFactors` very easy to read. Note that we used a `where` clause, but could equally have used a `let ...in`; in this case the choice is a matter of personal preference, though there are other situations where one or the other is required.

```
tdFactors :: Int -> [Int]
tdFactors n = facts n (2:[3,5..])
    where facts n (f:fs)
            | n < f * f      = [n]
            | n `mod` f == 0 =
                 f : facts (n `div` f) (f:fs)
            | otherwise      = facts n fs
```

As in C, we have gone as far as we can using native integers, and we'll switch at this point to big integers; note that Haskell has no long integers, so it's more restrictive than C. The switch is simpler for Haskell than for C, since big integers are provided directly in the language, in the `Integer` datatype. For the Miller-Rabin primality test, we first need to write the function to perform modular exponentiation, since Haskell doesn't provide one in any of its standard libraries:

```
powmod :: Integer -> Integer -> Integer -> Integer
powmod b e m =
    let times p q = (p*q) `mod` m
        pow b e x
            | e == 0    = x
            | even e    = pow (times b b)
                              (e `div` 2) x
            | otherwise = pow (times b b)
                              (e `div` 2) (times b x)
    in  pow b e 1
```

This function is rather more typical of Haskell than the `sieve` function that used arrays. The signature indicates that the function takes three `Integer` values and returns an `Integer` value. The function is written as if it is three functions because all functions in Haskell are curried, so `powmod` is actually a function that takes an integer b and returns a function that takes an integer e that returns a function that takes an integer m and returns an `Integer`; thus, it is only a colloquialism, and frankly wrong, to say that `powmod` is a function that takes three integers and returns an integer. The `let ...in` defines local values. Local function `times` performs modular multiplication mod m. Local function `pow` has three definitions, each with a guard (the predicate between the vertical bar `|` and the equal sign `=`); the expression corresponding to the first matching guard predicate is calculated and returned as the value of the function. Here, the first guard expression checks for termination and the other two expressions call the `pow` function recursively.

The strong pseudoprime test is implemented by three local functions in the `isSpsp` function: `getDandS` extracts the powers of 2 from $n - 1$, `spsp` takes the tuple returned by

`getDandS` as input and performs the easy-return test, and `doSpsp` computes and tests the powers of a. Note that `mod` and `div` are curried prefix functions; the back-quotes turn them into binary infix functions. Note also that `if` is an expression in Haskell, as opposed to a statement in imperative languages, which means that it returns a value instead of controlling program flow; thus there may be no else-less `if`, and both consequents of the `if` must have the same type.

```
isSpsp :: Integer -> Integer -> Bool
isSpsp n a =
    let getDandS d s =
            if even d then getDandS (d `div` 2) (s+1)
                      else (d, s)
        spsp (d, s) =
            let t = powmod a d n
            in  if t == 1 then True else doSpsp t s
        doSpsp t s
            | s == 0     = False
            | t == (n-1) = True
            | otherwise  = doSpsp ((t*t) `mod` n) (s-1)
    in  spsp $ getDandS (n-1) 0
```

Haskell makes it difficult to work with random numbers (it's possible, though inconvenient, in the same way that arrays were inconvenient in the Sieve of Eratosthenes) because they require a state to be maintained from one call to the next, so we use the primes less than a hundred as the bases for the Miller-Rabin primality test.

```
isPrime :: Integer -> Bool
isPrime n =
    let ps = [2,3,5,7,11,13,17,19,23,29,31,37,41,
              43,47,53,59,61,67,71,73,79,83,89,97]
    in  n `elem` ps || all (isSpsp n) ps
```

The `rhoFactor` function finds a single factor by the rho method. Function `f` is the random-number generator. Function `fact` implements the tortoise-and-hare loop recursively; the computations in the `where` clause are done prior to the guard expressions in the body of the function even though they syntactically follow the guard expressions in the source code, which can be confusing if you are unaccustomed to it. The `rhoFactor` function is called recursively with a different constant for the random number generator if the loop falls into a cycle (the `d == n` clause) or finds a composite factor (the `otherwise` clause).

```
rhoFactor :: Integer -> Integer -> Integer
rhoFactor n c =
    let f x = (x*x+c) `mod` n
        fact t h
            | d == 1    = fact t' h'
            | d == n    = rhoFactor n (c+1)
            | isPrime d = d
            | otherwise = rhoFactor d (c+1)
            where
                t' = f t
                h' = f (f h)
                d = gcd (t' - h') n
    in  fact 2 2
```

Function `rhoFactors` calls `rhoFactor` repeatedly until it completes the factorization of n. The first two clauses extract factors of 2, the third clause tests primality of a remaining cofactor, and the fourth clause adjusts n and the list of factors after calling `rhoFactor`. We solved the problem of factoring a perfect power of 2 differently than in the C version of the program, stopping as soon as n is reduced to 2.

```
rhoFactors :: Integer -> [Integer]
rhoFactors n =
    let facts n
            | n == 2    = [2]
            | even n    = 2 : facts (n `div` 2)
            | isPrime n = [n]
            | otherwise = let f = rhoFactor n 1
                          in  f : facts (n `div` f)
    in  sort $ facts n
```

A main program that exercises the functions defined above is shown below. To compile the program with the GHC compiler, assuming it is stored in file `primes.hs`, say `ghc -o prime prime.hs`, and to run the program say `./prime`.

```
main = do
    print $ ancientPrimes 100
    print $ primes 100
    print $ length $ primes 1000000
    print $ tdPrime 716151937
    print $ tdFactors 8051
    print $ powmod 437 13 1741
    print $ isSpsp 2047 2
    print $ isPrime 600851475143
    print $ isPrime 2305843009213693951
    print $ rhoFactors 600851475143
```

The output is the same as the C version of the program, except for the reduced input to `tdFactors`.

Appendix: Java

Java is an object-oriented language, widely used, with a very large collection of libraries. Like Haskell, Java provides big integers, linked lists and bit arrays natively, so we can quickly jump in to the coding. The functions are shown below, but we leave it to you to package them into classes as you wish; in the sample code we put all the functions into class `Main`. We will be more careful here than in the two prior versions to ensure that we validate all the input arguments. We begin with the Sieve of Eratosthenes, which we limit to `int`s, but if you prefer a larger data type you are free to change it.

```
public static LinkedList sieve(int n)
{
    BitSet b = new BitSet(n);
    LinkedList ps = new LinkedList();

    b.set(0,n);

    for (int p=2; p<n; p++)
    {
        if (b.get(p))
        {
            ps.add(p);
            for (int i=p+p; i<n; i+=p)
            {
                b.clear(i);
            }
        }
    }

    return ps;
}
```

We used the built-in data types `BitSet` and `LinkedList`; indeed, it is one of the benefits of programming in Java that the standard libraries provide so much useful code. In the optimized version of the Sieve of Eratosthenes, we also use the built-in exception `IllegalArgumentException` instead of creating our own exception; it's easier, and just as clear.

```
public static LinkedList primes(int n)
{
    if (n < 2)
    {
        throw new IllegalArgumentException("must be greater than one");
    }

    int m = (n-1) / 2;
    BitSet b = new BitSet(m);
    b.set(0, m);

    int i = 0;
    int p = 3;
    LinkedList ps = new LinkedList();
    ps.add(2);

    while (p * p < n)
    {
        if (b.get(i))
        {
            ps.add(p);
            int j = 2*i*i + 6*i + 3;
            while (j < m)
            {
                b.clear(j);
                j = j + 2*i + 3;
            }
        }
        i += 1; p += 2;
    }

    while (i < m)
    {
        if (b.get(i))
        {
            ps.add(p);
        }
        i += 1; p += 2;
    }

    return ps;
}
```

Another of the libraries that Java provides is the `BigInteger` library, and we switch from normal integers to `BigInteger` for the rest of our functions; `int` is sufficient for the Sieve of Eratosthenes, because sieving with a large n produces too much output to be useful, but for the other functions `BigInteger` is definitely useful. The `tdPrime` function validates its input in the first `if`, checks for even numbers in the second `if` statement, and checks for odd divisors in the body of the `while`.

```
public static Boolean tdPrime(BigInteger n)
{
    BigInteger two = BigInteger.valueOf(2);

    if (n.compareTo(two) < 0)
    {
        throw new IllegalArgumentException("must be greater than one");
    }

    if (n.mod(two).equals(BigInteger.ZERO))
    {
        return n.equals(two);
    }

    BigInteger d = BigInteger.valueOf(3);

    while (d.multiply(d).compareTo(n) <= 0)
    {
        if (n.mod(d).equals(BigInteger.ZERO))
        {
            return false;
        }
        d = d.add(two);
    }

    return true;
}
```

The `tdFactors` function domain-checks the input, removes factors of 2, and, if the remaining cofactor is not 1, begins a loop over the odd numbers starting from 3, trying each odd number in turn until it finds factors and the remaining cofactor is greater than the square of the current factor. Note the comparison to 1 after all the factors of 2 are removed, which is necessary to prevent a factor of 1 from being added to the list of factors when n is a power of 2; we handled that differently in the Haskell version of the program. As with the GMP functions in C, the messiness of doing arithmetic by calling functions hides instead of writing expressions the underlying simplicity of the algorithm.

```
public static LinkedList tdFactors(BigInteger n)
{
    BigInteger two = BigInteger.valueOf(2);
    LinkedList fs = new LinkedList();

    if (n.compareTo(two) < 0)
    {
        throw new IllegalArgumentException("must be greater than one");
    }

    while (n.mod(two).equals(BigInteger.ZERO))
    {
        fs.add(two);
        n = n.divide(two);
    }

    if (n.compareTo(BigInteger.ONE) > 0)
    {
        BigInteger f = BigInteger.valueOf(3);
        while (f.multiply(f).compareTo(n) <= 0)
        {
            if (n.mod(f).equals(BigInteger.ZERO))
            {
                fs.add(f);
                n = n.divide(f);
            }
            else
            {
                f = f.add(two);
            }
        }
        fs.add(n);
    }

    return fs;
}
```

To a programmer not accustomed to object-oriented programming, it is annoying that the `add` method is overloaded, with the same method name referring to the addition of two `BigIntegers` when used as `f.add(two)` and to the insertion of an item in a `LinkedList` when used as `fs.add(f)`. Such usage may not be confusing to the compiler, because it keeps track of the types of all variables, but it can be confusing to the programmer who writes and reads the code and has to make sense of it.

The strong pseudoprime test in the `isSpsp` function computes d and s in the first `while` loop, checks for an early termination in the `if`, then counts down s in the second `while` loop. Note that the early termination test is different in the Java version than the Haskell version; the Haskell version separates the early termination test from the $n-1$ tests, but the Java version combines the early termination test with the first loop of the $n-1$ tests, then pre-decrements s before starting the second `while` loop. Both versions of the function get the right answer, so the choice is based on the convenience of the programmer.

```
private static Boolean isSpsp(BigInteger n, BigInteger a)
{
    BigInteger two = BigInteger.valueOf(2);
    BigInteger n1 = n.subtract(BigInteger.ONE);
    BigInteger d = n1;
    int s = 0;

    while (d.mod(two).equals(BigInteger.ZERO))
    {
        d = d.divide(two);
        s += 1;
    }

    BigInteger t = a.modPow(d, n);

    if (t.equals(BigInteger.ONE) || t.equals(n1))
    {
        return true;
    }

    while (--s > 0)
    {
        t = t.multiply(t).mod(n);
        if (t.equals(n1))
        {
            return true;
        }
    }

    return false;
}
```

After using a predefined list of bases in the Haskell version of the function, we're back to using random bases in the `isPrime` function. The two `if` tests check the input domain and exit quickly if the input is even, then the `while` loop performs 25 strong pseudoprime tests.

```
public static Boolean isPrime(BigInteger n)
{
    Random r = new Random();
    BigInteger two = BigInteger.valueOf(2);
    BigInteger n3 = n.subtract(BigInteger.valueOf(3));
    BigInteger a;
    int k = 25;

    if (n.compareTo(two) < 0)
    {
        return false;
    }

    if (n.mod(two).equals(BigInteger.ZERO))
    {
        return n.equals(two);
    }

    while (k > 0)
    {
        a = new BigInteger(n.bitLength(), r).add(two);
        while (a.compareTo(n) >= 0)
        {
            a = new BigInteger(n.bitLength(), r).add(two);
        }

        if (! isSpsp(n, a))
        {
            return false;
        }

        k -= 1;
    }

    return true;
}
```

Note that Java's `BigInteger` library includes a function `isProbablePrime` that performs this computation in exactly the same way.

The `rhoFactor` function races the tortoise and hare until the gcd is greater than 1, then the `if-else` chain either returns a prime factor or retries the factorization with a different random function.

```
private static BigInteger rhoFactor(BigInteger n, BigInteger c)
{
    BigInteger t = BigInteger.valueOf(2);
    BigInteger h = BigInteger.valueOf(2);
    BigInteger d = BigInteger.ONE;

    while (d.equals(BigInteger.ONE))
    {
        t = t.multiply(t).add(c).mod(n);
        h = h.multiply(h).add(c).mod(n);
        h = h.multiply(h).add(c).mod(n);
        d = t.subtract(h).gcd(n);
    }

    if (d.equals(n)) /* cycle */
    {
        return rhoFactor(n, c.add(BigInteger.ONE));
    }
    else if (isPrime(d)) /* success */
    {
        return d;
    }
    else /* found composite factor */
    {
        return rhoFactor(d, c.add(BigInteger.ONE));
    }
}
```

The `rhoFactors` function first validates its input, then extracts factors of 2 in the first `while`, and, unless the input is a power of 2, calls `rhoFactor` repeatedly in the second `while` until the remaining cofactor is prime, sorting the list of factors before returning it. The built-in `isProbablePrime` function is called rather than the one we defined above.

```
public static LinkedList rhoFactors(BigInteger n)
{
    BigInteger f;
    BigInteger two = BigInteger.valueOf(2);
    LinkedList fs = new LinkedList();

    if (n.compareTo(two) < 0)
    {
        return fs;
    }

    while (n.mod(two).equals(BigInteger.ZERO))
    {
        n = n.divide(two);
        fs.add(two);
    }

    if (n.equals(BigInteger.ONE))
    {
        return fs;
    }

    while (! n.isProbablePrime(25))
    {
        f = rhoFactor(n, BigInteger.ONE);
        n = n.divide(f);
        fs.add(f);
    }

    fs.add(n);
    Collections.sort(fs);
    return fs;
}
```

To show examples of the use of these functions, we have to create a complete program with all of its imports and a class declaration. The program shown below is decidedly simple-minded, sufficient only to show a few examples; you will surely want to arrange the class differently in your own programs. For sake of brevity, the function bodies are elided below.

```
import java.util.LinkedList;
import java.util.BitSet;
import java.util.Random;
import java.util.Collections;
import java.math.BigInteger;
import java.lang.Exception;
import java.lang.Boolean;

class Main {

    public static LinkedList sieve(int n) { ... }
    public static LinkedList primes(int n) { ... }
    public static Boolean tdPrime(BigInteger n) { ... }
    public static LinkedList tdFactors(BigInteger n) { ... }
    private static Boolean isSpsp(BigInteger n, BigInteger a) { ... }
    public static Boolean isPrime(BigInteger n) { ... }
    private static BigInteger rhoFactor(BigInteger n, BigInteger c) { ... }
    public static LinkedList rhoFactors(BigInteger n) { ... }

    public static void main (String[] args)
    {
        System.out.println(sieve(100));
        System.out.println(primes(100));
        System.out.println(primes(1000000).size());
        System.out.println(tdPrime(new BigInteger("600851475143")));
        System.out.println(tdFactors(new BigInteger("600851475143")));
        System.out.println(isPrime(new BigInteger("600851475143")));
        System.out.println(isPrime(new BigInteger("2305843009213693951")));
        System.out.println(rhoFactors(new BigInteger("600851475143")));
    }

}
```

Output from the program is the same as all the other implementations.

Appendix: Python

Python is a commonly-used scripting language with a reputation of being easy to read, write and learn, a mixed imperative/object-oriented flavor, and a large library, both standard and user-contributed. We'll take the opportunity with Python to extend the domain of integer factorization beyond the integers greater than 1 that mathematicians generally consider. Specifically, in the `rho_factors` function we'll consider -1, 0 and 1 to be prime, so they factor as themselves, and we'll factor negative numbers by adding -1 to the list of factors of the corresponding positive number. This isn't entirely correct, but it isn't entirely incorrect, either, and is actually useful in some cases. And we're in good company; Wolfram|Alpha calculates factors the same way we do.

We begin, as we did with Haskell, with a one-liner (folded here onto three lines to make it fit the narrow column, but with Python's offside rule about code indenting, it must all be presented to the interpreter on a single line) that purports to be the Sieve of Eratosthenes:

```
print [x for x in range(2,100)
  if not [y for y in range(2, int(x**0.5)+1)
    if x%y == 0]]
```

That prints the primes less than a hundred. But the expression `if x%y == 0` at the end gives the game away: it's really trial division (the `%` operator for modulo), so it's not a sieve. In fact, it is very slow as n grows large, about $O(n^2/2(log_e n)^2$. Don't be fooled by cute one-liners!

The ancient Sieve of Eratosthenes consists of two `for`-loops, the first over the integers 2 to n and the second over the multiples of the current prime. The bitarray `b` has $n+1$ elements, but `b[0]` and `b[1]` are never accessed; it's easier to add two extra bits than to map between two different representations of the same thing.

```
def sieve(n):
    b, p, ps = [True] * (n+1), 2, []
    for p in xrange(2, n+1):
        if b[p]:
            ps.append(p)
            for i in xrange(p, n+1, p):
                b[i] = False
    return ps
```

Next is the optimized Sieve of Eratosthenes. After validating the input, the first `while` loop collects the sieving primes and performs the sieving, and the second `while` loop collects the remaining primes that survived the sieve. Note that `append` adds an element after the target, unlike the C function that inserts an element before the target.

```
def primes(n):
    if type(n) != int and type(n) != long:
        raise TypeError('must be integer')
    if n < 2:
        raise ValueError('must be greater than one')
    m = (n-1) // 2
    b = [True] * m
    i, p, ps = 0, 3, [2]
    while p*p < n:
        if b[i]:
            ps.append(p)
            j = 2*i*i + 6*i + 3
            while j < m:
                b[j] = False
                j = j + 2*i + 3
        i += 1; p += 2
    while i < m:
        if b[i]:
            ps.append(p)
        i += 1; p += 2
    return ps
```

The next function, `td_prime`, has three possible return values: `PRIME`, `COMPOSITE`, and `UNKNOWN` when the limit is exceeded. We use `True` and `False` for `PRIME` and `COMPOSITE`, and raise an `OverflowError` exception if the limit is exceeded, requiring some effort for the user to trap the error and respond accordingly. After validating the input, the `if` checks even numbers and the `while` loop checks odd numbers. Python's optional-argument syntax is simple and convenient; the argument `limit` is optional, and is given a default value if not specified.

```
def td_prime(n, limit=1000000):
    if type(n) != int and type(n) != long:
        raise TypeError('must be integer')
    if n % 2 == 0:
        return n == 2
    d = 3
    while d * d <= n:
        if limit < d:
            raise OverflowError('limit exceeded')
        if n % d == 0:
            return False
        d += 2
    return True
```

The `td_factors` function has the same problem with the `limit` argument as `td_prime`, and we solve it the same way, raising an `OverflowError` exception if the `limit` is exceeded. The `if` becomes a `while` on the factors of 2, and the function collects factors in a list instead of returning as soon as it finds the first factor, but otherwise `td_factors` is very similar to `td_prime`.

```
def td_factors(n, limit=1000000):
    if type(n) != int and type(n) != long:
        raise TypeError('must be integer')
    fs = []
    while n % 2 == 0:
        fs += [2]
        n /= 2
    if n == 1:
        return fs
    f = 3
    while f * f <= n:
        if limit < f:
            raise OverflowError('limit exceeded')
        if n % f == 0:
            fs += [f]
            n /= f
        else:
            f += 2
    return fs + [n]
```

The Miller-Rabin primality checker makes use of Python's ability to nest functions to hide the strong pseudoprime checker inside the `is_prime` function. It also uses the built-in modular exponentiation function; with two arguments, `pow(b,e)` computes b^e, but with three arguments, `pow(b,e,m)` computes $b^e \pmod m$. Python offers random numbers, but we prefer to test a fixed set of bases.

```
def is_prime(n):
    if type(n) != int and type(n) != long:
        raise TypeError('must be integer')
    if n < 2:
        return False
    ps = [2,3,5,7,11,13,17,19,23,29,31,37,41,
          43,47,53,59,61,67,71,73,79,83,89,97]
    def is_spsp(n, a):
        d, s = n-1, 0
        while d%2 == 0:
            d /= 2; s += 1
        t = pow(a,d,n)
        if t == 1:
            return True
        while s > 0:
            if t == n-1:
                return True
            t = (t*t) % n
            s -= 1
        return False
    if n in ps: return True
    for p in ps:
        if not is_spsp(n,p):
            return False
    return True
```

Like `is_prime`, the `rho_factors` function reduces namespace pollution by hiding local functions; notice the `lambda`, which is an alternate way of creating a local function. Again, as with Java, the `+` function is overloaded for both addition and list construction. The code is similar to Java, even if it looks quite different.

```
def rho_factors(n, limit=1000000):
    if type(n) != int and type(n) != long:
        raise TypeError('must be integer')
    def gcd(a,b):
        while b: a, b = b, a%b
        return abs(a)
    def rho_factor(n, c, limit):
        f = lambda(x): (x*x+c) % n
        t, h, d = 2, 2, 1
        while d == 1:
            if limit == 0:
                raise OverflowError('limit exceeded')
            t = f(t); h = f(f(h)); d = gcd(t-h, n)
        if d == n:
            return rho_factor(n, c+1, limit)
        if is_prime(d):
            return d
        return rho_factor(d, c+1, limit)
    if -1 <= n <= 1: return [n]
    if n < -1: return [-1] + rho_factors(-n, limit)
    fs = []
    while n % 2 == 0:
        n = n // 2; fs = fs + [2]
    if n == 1: return fs
    while not is_prime(n):
        f = rho_factor(n, 1, limit)
        n = n / f
        fs = fs + [f]
    return sorted(fs + [n])
```

We could import the `gcd` function from the `fractions` library, but instead we implement it ourselves because it gives us the chance to discuss this famous algorithm. Donald E. Knuth, in Volume 2, Section 4.5.2 of his book *The Art of Computer Programming*, calls this the "granddaddy" of all algorithms because it is the oldest nontrivial algorithm that has survived to the present day. The algorithm is commonly called the Euclidean algorithm because it was described in Book VII, Propositions 1 and 2 of Euclid's *Elements*, but scholars believe the algorithm dates to about two hundred years before Euclid, sometime around 500 B.C. Knuth gives the entire history of the algorithm, and an extensive analysis of its time complexity, which is well worth your time. Euclid's version of the algorithm worked by repeatedly subtracting the smaller amount from the larger until they are the same; the modern version of the algorithm replaces subtraction with division (the modulo operator).

Here are some sample calls to the functions defined above; the answers are the same as all the other implementations.

```
print primes(100)
print len(primes(1000000))
print td_prime(600851475143)
print td_factors(600851475143)
print is_prime(600851475143)
print is_prime(2305843009213693951)
print rho_factors(600851475143)
```

Appendix: Scheme

Scheme is primarily an academic language, useful for expressing algorithms in imperative, functional, and message-passing styles, with a fully-parenthesized prefix syntax derived from Lisp. Scheme provides big integers natively, and also lists, but has no bit arrays, so our implementation of the ancient Sieve of Eratosthenes uses a vector of booleans, which uses eight bits per element instead of one but works perfectly well.

```
(define (sieve n)
    (let ((bits (make-vector (+ n 1) #t)))
      (let loop ((p 2) (ps '()))
        (cond ((< n p) (reverse ps))
              ((vector-ref bits p)
                (do ((i (+ p p) (+ i p))) ((< n i))
                  (vector-set! bits i #f))
                (loop (+ p 1) (cons p ps)))
              (else (loop (+ p 1) ps))))))
```

Let's take a moment for a quick lesson in Scheme. An expression like `(let ((var1 value1) ...) body)` establishes a local binding for each of several var/value pairs that is active in the body of the `let`; a `let*` expression is the same, except that the bindings are executed left-to-right, and each binding is available to those that follow.

Scheme has two looping constructs. The named-`let` variant of `let`, given by `(let name ((var1 value1) ...) body)`, is like `let`, but additionally binds *name* to a function with arguments var_k whose code is the body of the `let`, which executes `loop` when it is called recursively; by convention, the variable *name* is often called `loop`, but it is sometimes convenient to use other names.

The other looping construct is `do`, which is similar to the `for` of C. The form of the `do` loop is `(do ((var1 value1 next1) ...) (done? ret-value) body ...)`. Each var/value/next triplet specifies a variable *name*, a *value* for the variable when the `do` is initialized, and an *expression* evaluated at each step of the `do`; there may be multiple var/value/next triplets, in which case each is executed simultaneously, rather like a comma operator in a C `for` statement. The *done?* predicate terminates the `do` loop when it becomes true; this is the opposite of a C `for` loop, which terminates when the condition becomes false. The *return* value is optional; if it is not given, the return value of the `do` loop is unspecified. The statements in the body of the `do` loop are optional, and are evaluated only for their side effects.

Scheme also provides two conditional constructs. The first is `(if cond then else)`, which first evaluates the condition then evaluates one of the two succeeding clauses; like Haskell, an `if` is an expression, not a control-flow statement, but unlike Haskell, the `else` may be omitted, in which case the value of the `if` expression is undefined if the condition is false. `Cond` is similar to a nested set of `if` statements; each clause consists of a condition and body, the conditions are read in order until one is true, when the corresponding body is evaluated as the value of the `cond`.

Now that we know something about Scheme, we look at the optimized version of the Sieve of Eratosthenes:

```
(define (primes n)
  (if (or (not (integer? n)) (< n 2))
      (error 'primes "must be integer greater than one")
      (let* ((len (quotient (- n 1) 2))
             (bits (make-vector len #t)))
        (let loop ((i 0) (p 3) (ps (list 2)))
          (cond ((< n (* p p))
                 (do ((i i (+ i 1)) (p p (+ p 2))
                      (ps ps (if (vector-ref bits i) (cons p ps) ps)))
                     ((= i len) (reverse ps))))
                ((vector-ref bits i)
                  (do ((j (+ (* 2 i i) (* 6 i) 3) (+ j p)))
                      ((<= len j) (loop (+ i 1) (+ p 2) (cons p ps)))
                    (vector-set! bits j #f)))
                (else (loop (+ i 1) (+ p 2) ps)))))))
```

In the example above, *len* is the length of the bitarray, called m in the description of the algorithm, and *bits* is the bitarray itself, a vector of booleans. The `cond` has three clauses. The first clause is actually the termination clause of Step 5 and Step 6 that is executed last; the body-less `do` sweeps up the primes after sieving is complete, and the return value is the list of primes, which must be reversed because each newly-found prime is pushed to the front, not the back, of the accumulating list of primes. The second clause sifts each prime, as in Step 4; this `do` has a body, which clears the jth element of the bitarray, and the return value is an expression that calls the named-`let` recursively to advance to the next sieving prime. The `else` clause recurs when p is not prime.

Our `td-prime?` function follows the Scheme convention that predicates (functions that return a boolean) have names that end in a question mark. An error is signaled if *limit* is less than the smallest prime factor of n; this isn't as convenient as raising an exception in Python, because standard Scheme has no way to trap the error, but most implementations provide some kind of error trapping.

```
(define (td-prime? n . args)
  (if (or (not (integer? n)) (< n 2))
      (error 'td-prime? "must be integer greater than one")
      (let ((limit (if (pair? args) (car args) 1000000)))
        (if (even? n) (= n 2)
          (let loop ((d 3))
            (cond ((< limit d)
                    (error 'td-prime? "limit exceeded"))
                  ((< n (* d d)) #t)
                  ((zero? (modulo n d)) #f)
                  (else (loop (+ d 2)))))))))
```

`Td-factors` is similar to `td-prime?`, except that the first `if` becomes a loop, on *twos*, and both loops collect the factors that they find instead of stopping on the first factor. Here is a case where the names *twos* and *odds* in the named-`let` provide documentation of the nature of the loop, making the function clearer to the reader.

```
(define (td-factors n . args)
  (if (or (not (integer? n)) (< n 2))
      (error 'td-factors "must be integer greater than one")
      (let ((limit (if (pair? args) (car args) 1000000)))
        (let twos ((n n) (fs '()))
          (if (even? n)
              (twos (/ n 2) (cons 2 fs))
              (let odds ((n n) (d 3) (fs fs))
                (cond ((< limit d)
                        (error 'td-factors "limit exceeded"))
                      ((< n (* d d))
                        (reverse (cons n fs)))
                      ((zero? (modulo n d))
                        (odds (/ n d) d (cons d fs)))
                      (else (odds n (+ d 2) fs)))))))))
```

We've been using lists but haven't mentioned how they work. A list is either *null* or is a *pair* with an item in its `car` and another list in its `cdr`; the terms `car` and `cdr` are pre-historic. An item x is inserted at the front of a list *xs* by `(cons x xs)`; the word `cons` is short for *construct*. Predicates `(null? xs)` and `(pair? xs)` distinguish empty lists from non-empty ones. The null list is represented as `'()`, and the order of items in a list is reversed by `(reverse xs)`.

The function `prime?` that implements the Miller-Rabin primality checker illustrates two more features of Scheme. We have been introducing functions with the notation `(define (name args ...) body)`, but the alternate notation is `(define name (lambda (args ...) body))`. We use it here because we want variable *seed* to persist from one invocation of `prime?` to the next. Since the `let` is inside the `define` but outside the `lambda`, the variable bound by the `let` retains its value from one call of the function to the next, just like static variables in some programming languages. Thus, `prime?` is a closure, not just a function, because it encloses the *seed* variable. And while we're talking about `define`, even though it doesn't apply here, we have been using the dot-notation for some of our argument lists;

a construct like `(define (f args ... . rest) body)` provides a variable-arity argument list, with all arguments after the dot collected into a list *rest*.

The other Scheme feature is internal-`define`, which is used to provide local functions that don't pollute the global namespace. We define three local functions, `rand` that returns random numbers, `expm` that performs modular exponentiation (the name is a variant of `expt` that Scheme provides for the normal powering function) and `spsp?` that checks if a is a witness to the compositeness of n. And we're not done; the internal definition `expm` has its own internal definition `times` for modular multiplication. An internal-`define` must appear immediately after another `define`, a `lambda`, or a `let`.

```
(define prime?
  (let ((seed 3141592654))
    (lambda (n)
      (define (rand)
        (set! seed (modulo (+ (* 69069 seed) 1234567) 4294967296))
        (+ (quotient (* seed (- n 2)) 4294967296) 2))
      (define (expm b e m)
        (define (times x y) (modulo (* x y) m))
        (let loop ((b b) (e e) (r 1))
          (if (zero? e) r
            (loop (times b b) (quotient e 2)
                  (if (odd? e) (times b r) r)))))
      (define (spsp? n a)
        (do ((d (- n 1) (/ d 2)) (s 0 (+ s 1)))
            ((odd? d)
              (let ((t (expm a d n)))
                (if (or (= t 1) (= t (- n 1))) #t
                  (do ((s (- s 1) (- s 1))
                       (t (expm t 2 n) (expm t 2 n)))
                      ((or (zero? s) (= t (- n 1)))
                        (positive? s))))))))
      (if (not (integer? n))
          (error 'prime? "must be integer")
          (if (< n 2) #f
            (do ((a (rand) (rand)) (k 25 (- k 1)))
                ((or (zero? k) (not (spsp? n a)))
                  (zero? k))))))))
```

In the `prime?` function we define our own random number generator, since standard Scheme doesn't provide one; the static variable *seed* maintains the current state of the random number generator, which is of a type known as a linear-congruential generator. The multiplier 69069 is due to Knuth. Note that the seed is reset with each call to `rand`; the witness a is set to the range 1 to n, exclusive.

There are three `do` loops in the function. The first, the outer `do` in `spsp?`, binds two variables, d and s, and iterates until d is odd, performing Step 2 of Algorithm 4.A. The inner do in spsp? binds the two variables s and t and implements the loop of Step 4 and Step 5 of Algorithm 3.A, performing the modular squaring and terminating when s is zero or t is $n-1$. The result of the `do`-loop is computed by the predicate `(positive? s)`, which is `#t` if $t \equiv n-1 \pmod{n}$ and `#f` when s reaches 0 without finding $t \equiv n-1 \pmod{n}$. The `do` in the main body of the function uses the same idiom of having two terminating conditions and using the finishing predicate to differentiate the two.

Our final function implements integer factorization by Pollard's rho algorithm. The two internal definitions are `cons<`, which inserts an item into a list in ascending order instead of at the front, and `rho`, which implements the rho algorithm. The `cons<` function turns a vice into a virtue; since standard Scheme lacks a `sort` function, we insert the factors in order as we find them, instead of writing a sort function to sort them at the end, which gives the virtue of simpler code and is probably faster, given the short length of most lists of factors. The body of the function does error checking, extracts factors of 2, and assembles the complete factorization.

```
(define (rho-factors n . args)
  (define (cons< x xs)
    (cond ((null? xs) (list x))
          ((< x (car xs)) (cons x xs))
          (else (cons (car xs) (cons< x (cdr xs))))))
  (define (rho n limit)
    (let loop ((t 2) (h 2) (d 1) (c 1) (limit limit))
      (define (f x) (modulo (+ (* x x) c) n))
      (cond ((zero? limit) (error 'rho-factors "limit exceeded"))
            ((= d 1) (let ((t (f t)) (h (f (f h))))
                       (loop t h (gcd (- t h) n) c (- limit 1))))
            ((= d n) (loop 2 2 1 (+ c 1) (- limit 1)))
            ((prime? d) d)
            (else (rho d (- limit 1))))))
  (if (not (integer? n))
      (error 'rho-factors "must be integer")
      (let ((limit (if (pair? args) (car args) 1000)))
        (cond ((<= -1 n 1) (list n))
              ((negative? n) (cons -1 (rho-factors (- n) limit)))
              ((even? n)
                (if (= n 2) (list 2)
                  (cons 2 (rho-factors (/ n 2) limit))))
              (else (let loop ((n n) (fs '()))
                      (if (prime? n)
                          (cons< n fs)
                          (let ((f (rho n limit)))
                            (loop (/ n f) (cons< f fs)))))))))))
```

Here are some examples:

```
> (sieve 100)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
 79 83 89 97)
> (primes 100)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
 79 83 89 97)
> (length (primes 1000000))
78498
> (td-prime? 600851475143)
#f
> (td-factors 600851475143)
(71 839 1471 6857)
> (prime? 600851475143)
#f
> (prime? 2305843009213693951)
#t
> (rho-factors 600851475143)
(71 839 1471 6857)
```

As your reward for reading this far, we give two additional functions related to prime numbers: the segmented Sieve of Eratosthenes and a fast variant of Pollard's rho algorithm.

The basic idea of a segmented sieve is to choose the sieving primes less than the square root of n, choose a reasonably large segment size that nevertheless fits in memory, and then sieve each of the segments in turn, starting with the smallest. At the first segment, the smallest multiple of each sieving prime that is within the segment is calculated, then multiples of the sieving prime are marked as composite in the normal way; when all the sieving primes have been used, the remaining unmarked numbers in the segment are prime. Then, for the next segment, for each sieving prime you already know the first multiple in the current segment (it was the multiple that ended the sieving for that prime in the prior segment), so you sieve on each sieving prime, and so on until you are finished with all the segments.

As an example, we compute the primes from 100 to 200; we choose a segment size of 20, and the sieving primes are 3, 5, 7, 11 and 13. The first segment has the ten values {101 103 105 107 109 111 113 115 117 119}. The smallest multiple of 3 in the segment is 105, so strike 105 and each third number after: 111, 117. The smallest multiple of 5 in the segment is 105, so strike 105 and the fifth number after: 115. The smallest multiple of 7 in the segment is 105, so strike 105 and the seventh number after: 119. There is no multiple of 11 in the segment, so there is nothing to do. The smallest multiple of 13 in the segment is 117, so strike it. The numbers that are left {101 103 107 109 113} are prime. When resetting for the second segment {121 123 125 127 129 131 133 135 137 139} the smallest multiples of each sieving prime are 123, 125, 133, 121 and 143 (beyond the segment), which can all be calculated by counting the next multiple of the sieving prime after the end of the first segment.

Our function `seg-sieve` produces a list of primes from lo to hi; we assume that $lo > \sqrt{hi}$, and to keep things simple we also assume that both lo and hi are even. Variable ps is the list of sieving primes, initialized by the normal sieve, and qs is the offset of the smallest multiple of the corresponding sieving prime in the current segment. The segment size b is $\sqrt{hi}$, bs is the bitarray, zs is the output list of primes, and z is a function that adds a new prime to the list.

```
(define (seg-sieve lo hi)
  (let* ((r (inexact->exact (ceiling (sqrt hi))))
         (b (quotient r 2)) (bs (make-vector b #t))
         (ps (cdr (primes r)))
         (qs (map (lambda (p)
                    (modulo (* -1/2 (+ lo 1 p)) p)) ps))
         (zs (list)) (z (lambda (p) (set! zs (cons p zs)))))
    (do ((t lo (+ t b b))
         (qs qs (map (lambda (p q) (modulo (- q b) p))
                     ps qs)))
        ((<= hi t) (reverse zs))
      (when (< hi (+ t b b)) (set! b (quotient (- hi t) 2)))
      (do ((i 0 (+ i 1))) ((= i b)) (vector-set! bs i #t))
      (do ((ps ps (cdr ps)) (qs qs (cdr qs))) ((null? qs))
        (do ((j (car qs) (+ j (car ps)))) ((<= b j))
          (vector-set! bs j #f)))
      (do ((j 0 (+ j 1))) ((= j b))
        (if (vector-ref bs j) (z (+ t j j 1)))))))
```

The outer `do` iterates over the segments; the qs are initialized by mapping the smallest multiple of the sieving prime onto the bitarray that holds the segment and are reset from one segment to the next by adding the segment size modulo the sieving prime. The `when` resets the top of the segment the last time through when only a partial segment remains. The first inner `do` resets the bitarray, the second inner `do` sieves each prime, its inner `do` strikes the multiples, and the final `do` sweeps up the primes. Here's an example:

```
> (apply + (seg-sieve 1000000 2000000))
105363426899
```

For the fast variant of Pollard's rho algorithm, we make two changes to the basic version. The first change is algorithmic: we replace Floyd's turtle-and-hare cycle-finding algorithm, which Pollard used in his original version of the rho algorithm, with Brent's powers-of-two cycle-finding algorithm. Each time the step-counter i is a power of two, the value of x_i is saved; if a subsequent $_j = x_i$ is found before $j = 2i$, a cycle has been identified. Brent's cycle-finding algorithm requires only one modular multiplication per step, instead of the three modular multiplications required by Floyd's cycle-finding algorithm, so even though Brent's method typically requires more steps that Floyd's method, in practice the number of modular multiplications is generally about a quarter less than Floyd's method, giving a welcome speed-up to Pollard's factoring algorithm.

For instance, consider the cyclical sequence 1, 2, 3, 4, 5, 6, 3, 4, 5, 6, ... Initially $j = 1$, $x_j = 1$, $q = 2j = 2$ and the saved $x = 1$. Then $j = 2$, $x_j = 2$, q is reset to 4 and the saved x is reset to 2. Then $j = 3$, then $j = 4$, and q is reset to 8 and the saved x is reset to 4. This continues until $j = 8$ and $x_j = 4$, which equals the saved x, identifying the cycle.

The second change is code-tuning: we replace the gcd at each step with a gcd that is calculated only periodically, performing instead a modular multiplication at each step, which is much faster than a gcd calculation. This is done by taking the product of all the $|x_{i+1} - x_i|$ modulo n for several steps, then taking a gcd of the product at the end. If the gcd is 1, then all the intermediate gcd calculations were also 1. If the gcd is prime, it is a factor of n. If the gcd is composite (including the case where the gcd is equal to n) it is necessary to retreat to the saved value of x from the prior gcd calculation and proceed step-by-step through the gcd calculations. The number of steps between successive gcd calculations varies with the size of n (bigger n means less frequent gcd calculations) and the number of trial divisions performed before starting the rho algorithm (more trial divisions means less frequent gcd calculations); values between 10 and 250 may be appropriate depending on the circumstances.

```
(define (brent n c limit)
  (define (f y) (modulo (+ (* y y) c) n))
  (define (g p x y) (modulo (* p (abs (- x y))) n))
  (let loop1 ((x 2) (y (+ 4 c)) (z (+ 4 c)) (j 1) (q 2) (p 1))
    (if (= j limit) (error 'brent "timeout")
      (if (= x y) (brent n (+ c 1) (- limit j)) ; cycle
        (if (= j q) (let ((t (f y)))
            (loop1 y (f y) z (+ j 1) (* q 2) (g p y t)))
            (if (positive? (modulo j 25))
                (loop1 x (f y) z (+ j 1) q (g p x y))
                (let ((d (gcd p n)))
                  (if (= d 1) (loop1 x (f y) y (+ j 1) q (g p x y))
                    (if (< 1 d n) d ; factor
                      (let loop2 ((k 1) (z (f z)))
                        (if (= k 25) (brent n (+ c 1) (- limit j))
                          (let ((d (gcd (- z x) n)))
                            (if (= d 1) (loop2 (+ k 1) (f z))
                              (if (= d n) (brent n (+ c 1) (- limit j))
                                d)))))))))))))
```

Our function keeps three values of the random-number sequence: x is the running value, y is the value at the last gcd calculation, and z is the value at the last power of two. The main loop also defines j as the step counter, q as the next power of two, and p as the current product for the short-circuit gcd calculation. Function f delivers the next value in the random-number sequence and function g accumulates the product of the differences between successive values of the sequence for the short-circuit gcd calculation. `Loop1` is the main body of the function, and `loop2` reruns the short-circuit gcd calculation when necessary; both call the function recursively if they find a cycle. We arbitrarily choose 25 as the number of steps between successive gcd calculations.