Computation of $\pi(n)$ by Sieving, Primality Testing, Legendre's Formula and Meissel's Formula

Jason Eisner, Spring 1993

This was one of several optional small computational projects assigned to undergraduate mathematics students at Cambridge University in 1993. I'm releasing my code and writeup in 2002 in case they are helpful to anyone—someone doing research in this area wrote to me asking for them.

My linear-time version of the Sieve of Eratosthenes may be original; I have not seen that algorithm anywhere else. But the rest of this work is straightforward implementation and exposition of well-known methods. A good reference is H. Riesel, Prime Numbers and Computer Methods for Factorization*.*

My Common Lisp implementation is in the file `primes.lisp`*. The standard language reference (now available online for free) is Guy L. Steele, Jr.,* Common Lisp: The Language*, 2nd ed., Digital Press, 1990.*

Note: In my discussion of running time, I have adopted the usual idealization of a machine that can perform addition and multiplication operations in constant time. Real computers obviously fall short of this ideal; for example, when n and m are represented in base 2 by arbitrary length bitstrings, it takes time $O(\log n \log m)$ to compute nm.

Introduction: In this project we'll look at several approaches for finding $\pi(n)$, the numberof primes less than n. Each approach has its advantages.

- *Sieving* produces a complete list of primes that can be further analyzed. For instance, after sieving, we may easily identify the 8169 pairs of twin primes below 10^6.
- *Legendre's method* and its variants need much less memory than sieves. However, they return less information and are in general somewhat slower as $n \to \infty$.

- Some modern *primality-testing* methods allow us to detect very large primes efficiently. They are not ideal for counting all the primes in [1,n], but for other tasks—like finding the first prime beyond 10^{100}—they are just about the only option.

Results: I'll get these out of the way up front. Lisp printed the results below as a single table. I have taken the liberty of splitting that verbatim output into two parts, to fit this narrow page.

```
> (tabulate-pi-methods)
            Pi-Tested-  Pi-Tested   Pi-Tested-  Pi-Tested-  Pi-Sieved
            Slow                   Probably    Grh

10          4           4           4           4           4
20          8           8           8           8           8
30          10          10          10          10          10
40          12          12          12          12          12
50          15          15          15          15          15
60          17          17          17          17          17
70          19          19          19          19          19
80          22          22          22          22          22
90          24          24          24          24          24
100         25          25          25          25          25
200         46          46          46          46          46
300         62          62          62          62          62
400         78          78          78          78          78
500         95          95          95          95          95
600         109         109         109         109         109
700         125         125         125         125         125
800         139         139         139         139         139
900         154         154         154         154         154
1000        168         168         168         168         168
2000        303         303         303         303         303
3000        430         430         430         430         430
4000        550         550         550         550         550
5000        669         669         669         669         669
6000        783         783         783         783         783
7000        900         900         900         900         900
8000        1007        1007        1007        1007        1007
9000        1117        1117        1117        1117        1117
10000       1229        1229        1229        1229        1229
20000       2262        2262                                2262
30000       3245        3245                                3245
40000       4203        4203                                4203
50000       5133        5133                                5133
60000       6057        6057                                6057
70000       6935        6935                                6935
80000       7837        7837                                7837
90000       8713        8713                                8713
100000      9592        9592                                9592
200000                                                      17984
300000                                                      25997
400000                                                      33860
```

500000	41538
600000	
700000	
800000	
900000	
1000000	
2000000	
3000000	
4000000	
5000000	
6000000	
7000000	
8000000	
9000000	
10000000	

	Pi-Sieved-Linear	Pi-Sieved-Linear-Fast	Pi-Legendre	Pi-Legendre-Fast	Pi-Meissel
10	4	4	4	4	4
20	8	8	8	8	8
30	10	10	10	10	10
40	12	12	12	12	12
50	15	15	15	15	15
60	17	17	17	17	17
70	19	19	19	19	19
80	22	22	22	22	22
90	24	24	24	24	24
100	25	25	25	25	25
200	46	46	46	46	46
300	62	62	62	62	62
400	78	78	78	78	78
500	95	95	95	95	95
600	109	109	109	109	109
700	125	125	125	125	125
800	139	139	139	139	139
900	154	154	154	154	154
1000	168	168	168	168	168
2000	303	303	303	303	303
3000	430	430	430	430	430
4000	550	550	550	550	550
5000	669	669	669	669	669
6000	783	783	783	783	783
7000	900	900	900	900	900
8000	1007	1007	1007	1007	1007
9000	1117	1117	1117	1117	1117
10000	1229	1229	1229	1229	1229
20000	2262	2262	2262	2262	2262
30000	3245	3245	3245	3245	3245
40000	4203	4203	4203	4203	4203
50000	5133	5133	5133	5133	5133
60000	6057	6057	6057	6057	6057
70000	6935	6935	6935	6935	6935
80000	7837	7837	7837	7837	7837

90000	8713	8713	8713	8713	8713
100000	9592	9592	9592	9592	9592
200000			17984	17984	17984
300000			25997	25997	25997
400000			33860	33860	33860
500000			41538	41538	41538
600000			49098	49098	49098
700000			56543	56543	56543
800000			63951	63951	63951
900000			71274	71274	71274
1000000			78498	78498	78498
2000000				148933	148933
3000000				216816	216816
4000000				283146	283146
5000000				348513	348513
6000000				412849	412849
7000000				476648	476648
8000000				539777	539777
9000000				602489	602489
10000000				664579	664579

The ten columns correspond to the following algorithms:

pi-tested-slow Almost, but not quite, the stupidest test imaginable. To test whether 91 is prime, it divides it by all integers from 2 up to $\sqrt{91}$ until it finds a divisor.

pi-tested A much more sensible version of `pi-tested-slow`, that only uses primes as divisors. This implementation is notable only for never taking a square root.

pi-tested-probably Uses the probabilistic primality test of Miller and Rabin. It may occasionally let a composite number through, but we can control the size of the needle's eye. The needle in the column above should have admitted at most one composite number in ten billion.

pi-tested-GRH Like `pi-tested-probably`, tests whether each integer is a strong pseudoprime to various bases. If the Generalized Riemann Hypothesis is true, the test in `pi-tested-GRH` identifies primes perfectly.

pi-sieved The classic sieve of Eratosthenes.

pi-sieved-linear My improved version of the sieve, which runs in theoretically linear time.

pi-sieved-linear-fast A constant-time speedup on pi-sieved-linear.

pi-Legendre Legendre's formula, in terms of the $\phi(x, a)$ function.

pi-Legendre-fast The suggested improvement to pi-Legendre.

pi-Meissel An efficient implementation of Meissel's formula.

An opening amusement: The shortest prime-testing program I can think of is (of course) in APL:

$$\wedge/0 \neq (1 + \iota N - 2)|N$$

determines whether N is prime. Similarly

$$(2 = +/[1]0 = V \circ .|V)/V \leftarrow \iota N$$

lists all the primes up to N. Unfortunately these algorithms are $O(N)$ and $O(N^2)$ respectively, in both time and space!

The division algorithm: A much better way to test whether n is prime is to divide it by already identified smaller primes $2, 3, 5, \ldots$ until a prime divisor is found. If there is no prime divisor $\leq \sqrt{n}$, then n must be prime.

Running time of the division algorithm: How long does it take to run this division test on $1, 2, 3 \ldots n$? All the numbers must be tested for divisibility by 2; the ones indivisible by 2 must be tested again; the ones indivisible by 2 and 3 must be tested yet again; and so on. So the algorithm takes time proportional to

$$\begin{aligned}
& \sum_{p \leq \sqrt{n}} \text{number of integers in } [1, n] \text{ tested for } p\text{-divisibility} \\
= & \sum_{p \leq \sqrt{n}} \text{number of integers in } [1, n] \text{ indivisible by primes} < p \\
\approx & \sum_{p \leq \sqrt{n}} n \cdot (1 - 1/2) \cdot (1 - 1/3) \cdot (1 - 1/5) \cdots (1 - 1/p') \\
= & \sum_{p \leq \sqrt{n}} n \cdot r(p') \text{ say,}
\end{aligned}$$

where p' is the prime preceding p.[1]

[1]The Chinese Remainder Theorem says that for any values of $(a_2, a_3, a_5, \ldots a_{p'})$, the range $[1, 2 \cdot 3 \cdot 5 \cdots p']$ contains exactly one integer whose residue classes mod $(2, 3, 5, \ldots p')$ are the a_i. It follows that the range contains exactly $(2-1)(3-1)(5-1)\cdots(p'-1)$ integers whose residue classes are all nonzero. So $r(p') = \cdot(2-1)(3-1)(5-1)\cdots(p'-1)/(2 \cdot 3 \cdot 5 \cdots p')$ is actually the proportion of integers in *this* range (rather than $[1, n]$) that are indivisible by all primes $< p$.

Recall the theorem in Part II Number Theory that $1/r(p) \geq \log(p+1)$. This tells us that $r(p) \leq 1/\log p$. Indeed, $p^2 r(p)$ approximates $\pi(p^2)$, which is the number of integers in $[1, p^2]$ that are indivisible by primes $< p$; so we expect $r(p) \approx \pi(p^2)/p^2 \approx 1/(2\log p)$.

So if q is the greatest prime below $\sqrt{n}$, the time complexity is about

$$\begin{aligned} n \sum_{p \leq \sqrt{n}} r(p') &= n(1 - r(q) + \sum_{p \leq \sqrt{n}} r(p)) \\ &\approx n \sum_{p \leq \sqrt{n}} r(p) \\ &\approx n \sum_{p \leq \sqrt{n}} \frac{1}{2\log p}. \end{aligned}$$

Let's put some bounds on this. On the one hand,

$$\begin{aligned} n \sum_{p \leq \sqrt{n}} \frac{1}{2\log p} &\geq n \sum_{p \leq \sqrt{n}} \frac{1}{2p} \\ &= O(n \log\log \sqrt{n}) \\ &= O(n \log\log n). \end{aligned}$$

On the other,

$$\begin{aligned} n \sum_{p \leq \sqrt{n}} \frac{1}{2\log p} &\leq n \sum_{2 \leq i \leq \sqrt{n}} \frac{1}{2\log i} \\ &\leq n \sum_{2 \leq i \leq \sqrt{n}} \frac{1}{2\log i} \\ &\leq n \int_1^{\sqrt{n}} \frac{dx}{2\log x} \\ &\leq n(\int_1^{e^2} \frac{dx}{2\log x} + \int_{e^2}^{\sqrt{n}} (\frac{1}{\log x} - \frac{1}{(\log x)^2}) dx) \\ &= n(\text{const} + \frac{\sqrt{n}}{\log \sqrt{n}}) = O(\frac{n^{3/2}}{\log n}). \end{aligned}$$

In summary, the division algorithm finds $\pi(n)$ in time

$$O(n \cdot \sum_{p \leq \sqrt{n}} 1/\log p),$$

which expression is somewhere between $O(n \cdot \log\log n)$ and $O(n \cdot \sqrt{n}/\log n)$.

Space complexity of the division algorithm: The division method requires only $\pi(n) = O(n/\log n)$ space to list the primes up to n; the space is used to hold the primes as they are found. Moreover, if we merely want to compute $\pi(n)$, then we can get away with $\pi(\sqrt{n}) = O(\sqrt{n}/\log n)$—just enough space to hold the primes being used as *divisors*. (This is what my implementation does.)

Complexity of the Sieve of Eratosthenes: The standard Sieve of Eratosthenes, applied to the range [1,n], requires $O(n)$ space. It is dominated by an inner loop that strikes out one composite number for every nontrivial multiple (below n) of each prime (below $\sqrt{n}$). Thus it requires time on the order of $\sum_{p \le \sqrt{n}} \left\lfloor \frac{n}{p} - 1 \right\rfloor$. This exceeds

$$\begin{aligned} \sum_{p \le \sqrt{n}} (\frac{n}{p} - 2) &> (\sum_{p \le \sqrt{n}} \frac{n}{p}) - 2n \\ &= n((\sum_{p \le \sqrt{n}} \frac{1}{p}) - 2) \end{aligned}$$

and is exceeded by

$$\sum_{p \le \sqrt{n}} \frac{n}{p} = n \sum_{p \le \sqrt{n}} \frac{1}{p},$$

so is $O(n \log\log n)$ by a result in Part II Number Theory. In other words, the sieve does at least as well as the division method.

Comparing the sieve and division methods: The standard sieve improves upon the division method by not even attempting to divide odd numbers by 2, or 3-free numbers by 3. It skips right over them. On the other hand, unlike the division method, the sieve method will repeatedly strike out a number with many distinct prime factors. Every multiple of 6 is struck at least twice.

More concisely: The division method spends disproportionate time on numbers with *no small* prime factors. The sieve spends disproportionate time on numbers with *many* prime factors.

A better sieve: As it turns out, however, it is actually possible to design a less wasteful sieve that catches each composite number exactly once, and still spends no time on the prime numbers. Such a sieve will take only *linear* time. (The Prime Number Theorem shows that [1,n] contains $O(n - n/\log n) \sim O(n)$ composite numbers.)

The trick is to strike out integers whose *smallest* (prime) factor is 2, then those whose smallest factor is 3, then 5, 7, etc. How do we arrange this? Observe (for example) that $k \leq n$ has 7 as its least factor iff $k = 7 \cdot k'$ for some 2,3,5-free integer $k' \leq n/7$—i.e., some integer not struck out of the sieve on a previous pass.

So on the 7 pass, we merely need to multiply by 7 those surviving elements of the sieve that are $\leq n/7$, and strike the products out of the sieve. Three straightforward tricks are necessary:

- Our list of multiplicands is the sieve itself, which we are in the process of destroying. To avoid striking out sieve elements before we have used them as multiplicands, we must strike them in decreasing order. For example, we strike $7 \cdot 91$ *before* we strike $7 \cdot 13 = 91$.

- To ensure the linearity of the algorithm, we want each element to be struck in constant time. We impose upon the sieve array the additional structure of a (doubly) linked list, so that we can move from each element to the next-smallest element in constant time. (The reverse links are needed so that we can carry out the deletions.)

- For each prime multiplier p, we must somehow locate the starting multiplicand $S(p)$, i.e., the largest integer $\leq n/p$ that remains in the sieve. We maintain a table of the values of $S(k)$ for $1 \leq k \leq \sqrt{n}$, updating it via backpointers as elements of the sieve are struck out.

  These updates take at worst $O(n)$ time: for we make $\pi(\sqrt{n}) \leq \sqrt{n}$ passes, and there are only $\sqrt{n}$ values of $S(k)$ that could possibly be updated on each pass.[2]

The resulting algorithm has the desired $O(n)$ complexity, in space as well as in time, and it strikes each composite number exactly once. (Indeed, my implementation of the algorithm simply calculates $\pi(n)$ as $(n-1)$ − (number of strikes).) This is a happy result: I know of no faster method to compute $\pi(n)$, let alone find all the primes that $\pi(n)$ counts.[3]

[2]There are several other ways to find $S(p)$. A very simple and convenient solution is to sweep *upwards* through potential multiplicands till we pass n/p, then work back down again to do the real work of multiplication and deletion. This only damages our running time by a constant factor. Even more simply, we might start at $\lfloor n/p \rfloor$ and decrement by 1 till we come upon a number still in the sieve; but it is by no means trivial that this last approach preserves the linearity of the algorithm!

[3]Presumably linear sieving algorithms have appeared in the literature before. However, I am unaware of any—if I have actually done something original here, please let me know.

I should caution that we obtain this $O(n)$ running time only in theory. We have been pretending that our computer performs multiplication in time $O(1)$. On a conventional machine, the multiplications will (I think) drive our time up to $O(n(\log n)^2)$. The standard sieve only uses addition, and so should take time only $O(n \log n \log\log n)$.

Legendre's formula: Slower methods to compute $\pi(n)$ may be more space-efficient. One such is Legendre's formula.

Computing $\pi(128)$ from Legendre's formula, without so much as a calculator to ease the burden:

Note that $\left\lfloor \sqrt{128} \right\rfloor = 11$.

$$\begin{aligned}
\pi(128) &= -1 + \pi(\sqrt{128}) + 128 \\
&\quad -(\left\lfloor \frac{128}{2} \right\rfloor + \left\lfloor \frac{128}{3} \right\rfloor + \left\lfloor \frac{128}{5} \right\rfloor + \left\lfloor \frac{128}{7} \right\rfloor + \left\lfloor \frac{128}{11} \right\rfloor) \\
&\quad +(\left\lfloor \frac{128}{6} \right\rfloor + \left\lfloor \frac{128}{10} \right\rfloor + \left\lfloor \frac{128}{15} \right\rfloor + \left\lfloor \frac{128}{14} \right\rfloor + \left\lfloor \frac{128}{21} \right\rfloor \\
&\qquad + \left\lfloor \frac{128}{35} \right\rfloor + \left\lfloor \frac{128}{22} \right\rfloor + \left\lfloor \frac{128}{33} \right\rfloor + \left\lfloor \frac{128}{55} \right\rfloor + \left\lfloor \frac{128}{77} \right\rfloor) \\
&\quad -(\left\lfloor \frac{128}{30} \right\rfloor + \left\lfloor \frac{128}{42} \right\rfloor + \left\lfloor \frac{128}{70} \right\rfloor + \left\lfloor \frac{128}{105} \right\rfloor + \left\lfloor \frac{128}{66} \right\rfloor + \left\lfloor \frac{128}{110} \right\rfloor) \\
&= -1 + \pi(\sqrt{128}) + 128 - (64 + 42 + 25 + 18 + 11) + \\
&\quad (21 + 12 + 8 + 9 + 6 + 3 + 5 + 3 + 2 + 1) \\
&\quad -(4 + 3 + 1 + 1 + 1 + 1) \\
&= -1 + \pi(\sqrt{128}) + 128 - 160 + 70 - 11 \\
&= 26 + \pi(\sqrt{128}).
\end{aligned}$$

But

$$\begin{aligned}
\pi(\sqrt{128}) &= \pi(11) = -1 + \pi(\sqrt{11}) + 11 - (\left\lfloor \frac{11}{2} \right\rfloor + \left\lfloor \frac{11}{3} \right\rfloor) + (\left\lfloor \frac{11}{6} \right\rfloor) \\
&= -1 + \pi(\sqrt{11}) + 11 - (5 + 3) + 1 \\
&= 3 + \pi(\sqrt{11}),
\end{aligned}$$

and

$$\pi(\sqrt{11}) = \pi(3) = -1 + \pi(1) + 3 = 2 + \pi(1).$$

Since $\pi(1) = 0$, we conclude that $\pi(128) = 31$.

A poor implementation of Legendre's formula: It is indeed easy to write a program mimicking the formula for ϕ, provided that we have a list of primes up to $\sqrt{x}$.[4] We can then compute $\pi(x)$ from $\phi(x, A)$, where

$$A = A(x) = \pi(\sqrt{x}) \approx \frac{2\sqrt{x}}{\log x}.$$

However, such a program is marvelously inefficient. A call of the form $\phi(\,\cdot\,, a)$ $(a > 0)$ spawns two calls of the form $\phi(\,\cdot\,, a-1)$. By induction, we see that it involves $2^{a+1}-1$ calls in total! Thus $\phi(x, A)$ requires $O(4^{\sqrt{x}/\log x})$ calls, worse than any polynomial algorithm.

A more sensible implementation of Legendre's formula: We are much better off if we add the extra stopping condition "$\phi(x, a) = 0$ when $x < 1$." In effect, this condition recognizes that many terms specified in Legendre's formula are zero: terms like $\left\lfloor \frac{128}{2 \cdot 7 \cdot 11} \right\rfloor$ may be omitted from the expansion above.

Time complexity of this algorithm: Given the extra stopping condition, what is the complexity of the algorithm to compute $\phi(x, a)$? First consider ϕ_k, defined by

$$\phi_k(x, a) = \begin{cases} 0 & \text{if } x < 1 \\ \lfloor x \rfloor & \text{if } a = 0 \\ \phi_k(x, a-1) - \phi_k(x/k, a-1) & \text{otherwise} \end{cases}$$

For convenience, we simply ignore all recursive calls for which $x < 1$. Each such call is paired with a call for which $x \geq 1$, so this artifice can affect the time complexity by a factor of 2 at most.[5]

Then $\phi(x, a)$ makes $\binom{b}{i}$ calls of the form $\phi_k(x/k^i, a-b)$, when $0 \leq b \leq a$ and $x \geq x/k^i \geq 1$. (As usual, $\binom{b}{i} = 0$ unless $0 \leq i \leq b$.) This gives

$$\sum_{(0 \leq b \leq a)} \sum_{(0 \leq i \leq \log_k x)} \binom{b}{i} = \sum_{(0 \leq i \leq \log_k x)} \sum_{(0 \leq b \leq a)} \binom{b}{i} = \sum_{0 \leq i \leq \log_k x} \binom{a+1}{i+1}$$

calls altogether.

[4] The project instructions suggest that we merely need to know $\pi(\sqrt{x})$, which is not enough.

[5] Alternatively, by putting an extra test in the code, we could ensure that such calls were never made.

Now simply observe that $\phi(x, A)$ takes time between $\phi_{\sqrt{x}}(x, A)$ and $\phi_2(x, A)$. We conclude that a lower bound, from $\phi_{\sqrt{x}}(x, A)$, is

$$\sum_{0 \le i \le 2} \binom{A+1}{i+1} = O(A) + O(A^2) + O(A^3) = O(A^3) = O(x^{3/2}/(\log x)^3).$$

(The corresponding upper bound, from $\phi_2(x, A)$, acquires an $O(A^{i+1})$ term for every i, as $x \to \infty$. So this upper bound is not very useful—it grows faster than any polynomial.)

Note in particular that the lower bound is slightly *worse* than linear; indeed, it is worse than $O(x \log\log x)$, the standard Sieve of Eratosthenes.[6] So both Legendre and division are slower than the sieve. (We have not settled the question of whether Legendre is faster or slower than division.)

Space requirements of Legendre's formula: By contrast, the space complexity is surprisingly low. It might appear that the $\phi(x, a)$ algorithm requires a stack of depth a, since $\phi(x, a)$ calls $\phi(x, a-1)$ calls ... calls $\phi(x, 0)$. This would imply a stack of depth about $\sqrt{x}/\log x$ for $\phi(x, A)$.

However, we can arrange for the $\phi(x, a-1)$ call to be tail-recursive! Thus only calls of the form $\phi(x/p_a, a-1)$ need to deepen the stack. Each such call diminishes the value of the first argument. It is not hard to see that the stack has a maximum depth of k, where k is the smallest integer with $x/m_k < 1$. So regardless of a, the stack size is at worst logarithmic in x:

$$(k > \log_2 x) \Rightarrow (m_k \ge 2^k > x) \Rightarrow (x/m_k < 1).$$

It follows that $\phi(x, A)$ requires no more than $O(\log x)$ space. This is far better than the other algorithms we have considered. (Remark: We have essentially used ϕ_2 to get an upper bound here.)

Cautionary remark: Note that the time and space complexities here are for computing $\pi(x) = \phi(x, A)$ once $p_1, p_2, \ldots p_A$ are known. The additional time and space required to compute and store the p_i, or to compute

[6] This hardly needs showing, but for the picky reader I suppose I can write

$$\begin{aligned}
O(e^y) &> O(y^8) \\
O(e^{\sqrt[8]{x}}) &> O(x) \\
O(\sqrt[8]{x}) &> O(\log x) \\
O(\sqrt{x}) &> O((\log x)^4) \\
O(\sqrt{x}) &> O((\log x)^3 (\log\log x)) \\
O(x^{3/2}/(\log x)^3) &> O(x \log\log x).
\end{aligned}$$

them on the fly, depends on the algorithm used to do so. For example, the sieve that I use here is fast ($O(\sqrt{n}\log\log n)$) but uses $O(\sqrt{n})$ space.

Caching results in Legendre's formula: Now consider the suggested optimization where m_k values of ϕ are precomputed and used many times in the recursion. We compute these values rapidly in time $O(m_k \log\log m_k)$ using a sieve, roughly as described in H. Riesel, *Prime Numbers and Computer Methods for Factorization*.

Assuming that m_k is substantially less than x, this optimization does save some time. Suppose the values of $\phi(t,k)$ are already known for $0 \le t \le m_k$. Then a call to $\phi(x,a)$ with the new algorithm takes no longer than a call to $\phi(x,a-k)$ with the old one.

Indeed, the only difference between the two calls is that we are dividing by primes $p_a, p_{a-1}, \ldots p_{a_{k+1}}$ instead of by the smaller primes $p_{a-k}, p_{a-k-1}, \ldots p_1$. This means that x may drop below 1 sooner in the new $\phi(x,a)$ than in the old $\phi(x,a-k)$—which means the algorithm finishes faster.

It is unlikely, however, that for fixed k this optimization actually reduces the complexity. A lower bound for $\phi(x)$ is still provided by $\phi_{\sqrt{x}}(x, A-k)$; this takes time $O((A-k)^3) = O(A^3) = O(x^{3/2}/(\log x)^3)$, just as for the original algorithm.

The space requirements increase by only a constant amount, $O(m_k)$, so are also unchanged in complexity.

Meissel's formula: Probably I ought to give a clear derivation of this. Borrowing some notation from Riesel's book, but clarifying the exposition, we write for any x, a:

$$\phi(x,a) = P_0(x,a) + P_1(x,a) + P_2(x,a) + \cdots$$

Here $P_k(x,a)$ is the number of products $\le x$ of exactly k primes $> p_a$. So the formula says that if a number $\le x$ is not divisible by any of the first a primes, then it is the product of some number of primes thereafter.

Note that $P_0(x,a) = 1$, and $P_1(x,a) =$ number of primes in $(p_a, x] = \pi(x) - a$. This is how $\pi(x)$ enters the formula.

Taking $b = \pi\sqrt{x}$, we can further write

$$\begin{aligned} P_2(x,a) &= \sum_{i>a} \#\text{ primes} \ge p_i \text{ whose products with } p_i \text{ are} \le x \\ &= \sum_{\pi(\sqrt{x})\ge i>a} (\#\text{ primes whose products with } p_i \text{ are} \le x \\ &\qquad\qquad - \#\text{ primes} < p_i) \end{aligned}$$

$$\begin{aligned} &= \sum_{b\geq i>a} (\pi(\frac{x}{p_i}) - (i-1)) \\ &= \left(\sum_{b\geq i>a} \pi(\frac{x}{p_i})\right) - \left(\sum_{i=1}^{b-1} i - \sum_{i=1}^{a-1} i\right) \\ &= \left(\sum_{b\geq i>a} \pi(\frac{x}{p_i})\right) - \frac{(b-a)(b+a-1)}{2}. \end{aligned}$$

If $p_{a+1} > \sqrt[3]{x}$—that is, if $a \geq c = \pi(\sqrt[3]{x})$—then a product of 3 or more primes $> p_a$ cannot be $\leq x$. Then $P_k(x,a) = 0$ for $k = 3, 4, 5 \ldots$.

So taking $a = c$, we obtain

$$\begin{aligned} \phi(x,c) &= P_0(x,c) + P_1(x,c) + P_2(x,c) \\ &= 1 + \pi(x) - c + (\sum_{b\geq i>c} \pi(\frac{x}{p_i})) - \frac{(b-c)(b+c-1)}{2} \\ &= \pi(x) + (\sum_{b\geq i>c} \pi(\frac{x}{p_i})) - \frac{(b+c-2)(b-c+1)}{2}, \end{aligned}$$

and solving for $\pi(x)$ gives us Meissel's formula.

Remark: It is worth noting that we did not have to choose $a = c$ above; any $a \geq c$ would have nullified P_3, P_4, etc. In particular, taking $a = b$ turns Meissel's formula into

$$\pi(x) = \phi(x,b) + b - 1,$$

which is exactly Legendre's formula.

If we had taken $a = \pi(\sqrt[4]{x})$, we would have had to deal with computing $P_3(x,a)$, but computing $\phi(x,a)$ would have been even easier.

Complexity of Meissel: The lovely thing about computing $\phi(x,c)$ is that the lower time bound from $\phi_{\sqrt{x}}(x,c)$ is only $c^3 \approx x/(\log x)^3$. Indeed, I conjecture that the computation of $\phi(x,c)$ really is linear or sublinear.

However, in Meissel's method we also have to compute π recursively for $O(\sqrt{x}/\log x)$ values less than $x^{2/3}$. Let us suppose that $\phi(x,c)$ really does take time $O(x)$ to compute (divided by some power of $\log x$). Then if $T(x)$ is the time to compute $\pi(x)$ with Meissel's formula, we can do this rough upper bound computation on the complexity:

$$\begin{aligned} T(x) &\leq O(x + \sqrt{x}\, T(x^{2/3})) \\ &\leq O(x + \sqrt{x}\,(x^{2/3} + x^{1/3} T(x^{4/9}))) \end{aligned}$$

$$\begin{aligned} &\leq O(x + \sqrt{x}\,(x^{2/3} + x^{1/3}(x^{4/9} + x^{2/9}(\cdots)))) \\ &= O(\sqrt{x}\,x^{2/3}) \\ &= O(x^{7/6}) \end{aligned}$$

divided by some power of $\log x$.

It is also necessary to generate the primes up to $\sqrt{x}$, of course, but this only need be done once; a sieve will take time sublinear in x. If we massage the results of the sieve a bit, we will be able to extract easily what we need to know: the values of $c = \pi(\sqrt[3]{x})$ and $b = \pi(\sqrt{x})$, the primes between 2 and c (for the call to ϕ), and the primes between c and b (for the summation). See the listing of `pi-array` for details.

Approximations to $\pi(x)$: Again, I have had to split one output table into two in order to fit it on the page. The first table gives the ratio of each approximation to the true value; the second gives the absolute error.

To see the original table, you can run `tabulate-approximations` yourself.

```
> (tabulate-approximations)
                     Pi     X/Log X          Li   Log-Ratio    Li-Ratio
10                4.000       4.343       6.165       0.921       0.649
20                8.000       6.676       9.905       1.198       0.808
30               10.000       8.820      13.022       1.134       0.768
40               12.000      10.843      15.839       1.107       0.758
50               15.000      12.781      18.469       1.174       0.812
60               17.000      14.654      20.965       1.160       0.811
70               19.000      16.476      23.362       1.153       0.813
80               22.000      18.256      25.678       1.205       0.857
90               24.000      20.001      27.930       1.200       0.859
100              25.000      21.715      30.126       1.151       0.830
200              46.000      37.748      50.192       1.219       0.916
300              62.000      52.597      68.333       1.179       0.907
400              78.000      66.762      85.418       1.168       0.913
500              95.000      80.456     101.794       1.181       0.933
600             109.000      93.795     117.646       1.162       0.927
700             125.000     106.853     133.089       1.170       0.939
800             139.000     119.678     148.197       1.161       0.938
900             154.000     132.306     163.023       1.164       0.945
1000            168.000     144.765     177.610       1.161       0.946
2000            303.000     263.127     314.809       1.152       0.962
3000            430.000     374.702     442.759       1.148       0.971
4000            550.000     482.273     565.364       1.140       0.973
5000            669.000     587.048     684.281       1.140       0.978
6000            783.000     689.694     800.414       1.135       0.978
7000            900.000     790.633     914.331       1.138       0.984
8000           1007.000     890.155    1026.416       1.131       0.981
9000           1117.000     988.470    1136.949       1.130       0.982
10000          1229.000    1085.736    1246.137       1.132       0.986
20000          2262.000    2019.491    2288.614       1.120       0.988
```

30000	3245.000	2910.092	3276.899	1.115	0.990
40000	4203.000	3774.783	4233.011	1.113	0.993
50000	5133.000	4621.167	5166.547	1.111	0.994
60000	6057.000	5453.504	6082.847	1.111	0.996
70000	6935.000	6274.510	6985.295	1.105	0.993
80000	7837.000	7086.054	7876.214	1.106	0.995
90000	8713.000	7889.501	8757.292	1.104	0.995
100000	9592.000	8685.890	9629.809	1.104	0.996
200000	17984.000	16385.287	18036.053	1.098	0.997
300000	25997.000	23787.741	26086.694	1.093	0.997
400000	33860.000	31009.627	33922.624	1.092	0.998
500000	41538.000	38102.892	41606.292	1.090	0.998
600000	49098.000	45096.897	49172.830	1.089	0.998
700000	56543.000	52010.443	56644.667	1.087	0.998
800000	63951.000	58856.563	64037.302	1.087	0.999
900000	71274.000	65644.796	71362.060	1.086	0.999
1000000	78498.000	72382.414	78627.555	1.084	0.998
2000000	148933.000	137848.727	149054.844	1.080	0.999
3000000	216816.000	201151.622	216970.580	1.078	0.999
4000000	283146.000	263126.650	283352.275	1.076	0.999
5000000	348513.000	324150.191	348638.140	1.075	1.000
6000000	412849.000	384436.227	413076.534	1.074	0.999
7000000	476648.000	444122.401	476826.847	1.073	1.000
8000000	539777.000	503304.442	539999.729	1.072	1.000
9000000	602489.000	562052.636	602676.298	1.072	1.000
10000000	664579.000	620420.688	664918.453	1.071	0.999

	Pi	X/Log X	Li	Log-Diff	Li-Diff
10	4.000	4.343	6.165	-0.343	-2.165
20	8.000	6.676	9.905	1.324	-1.905
30	10.000	8.820	13.022	1.180	-3.022
40	12.000	10.843	15.839	1.157	-3.839
50	15.000	12.781	18.469	2.219	-3.469
60	17.000	14.654	20.965	2.346	-3.965
70	19.000	16.476	23.362	2.524	-4.362
80	22.000	18.256	25.678	3.744	-3.678
90	24.000	20.001	27.930	3.999	-3.930
100	25.000	21.715	30.126	3.285	-5.126
200	46.000	37.748	50.192	8.252	-4.192
300	62.000	52.597	68.333	9.403	-6.333
400	78.000	66.762	85.418	11.238	-7.418
500	95.000	80.456	101.794	14.544	-6.794
600	109.000	93.795	117.646	15.205	-8.646
700	125.000	106.853	133.089	18.147	-8.089
800	139.000	119.678	148.197	19.322	-9.197
900	154.000	132.306	163.023	21.694	-9.023
1000	168.000	144.765	177.610	23.235	-9.610
2000	303.000	263.127	314.809	39.873	-11.809
3000	430.000	374.702	442.759	55.298	-12.759
4000	550.000	482.273	565.364	67.727	-15.364
5000	669.000	587.048	684.281	81.952	-15.281
6000	783.000	689.694	800.414	93.306	-17.414
7000	900.000	790.633	914.331	109.367	-14.331
8000	1007.000	890.155	1026.416	116.845	-19.416

9000	1117.000	988.470	1136.949	128.530	-19.949
10000	1229.000	1085.736	1246.137	143.264	-17.137
20000	2262.000	2019.491	2288.614	242.509	-26.614
30000	3245.000	2910.092	3276.899	334.908	-31.899
40000	4203.000	3774.783	4233.011	428.217	-30.011
50000	5133.000	4621.167	5166.547	511.833	-33.547
60000	6057.000	5453.504	6082.847	603.496	-25.847
70000	6935.000	6274.510	6985.295	660.490	-50.295
80000	7837.000	7086.054	7876.214	750.946	-39.214
90000	8713.000	7889.501	8757.292	823.499	-44.292
100000	9592.000	8685.890	9629.809	906.110	-37.809
200000	17984.000	16385.287	18036.053	1598.713	-52.053
300000	25997.000	23787.741	26086.694	2209.259	-89.694
400000	33860.000	31009.627	33922.624	2850.373	-62.624
500000	41538.000	38102.892	41606.292	3435.108	-68.292
600000	49098.000	45096.897	49172.830	4001.103	-74.830
700000	56543.000	52010.443	56644.667	4532.557	-101.667
800000	63951.000	58856.563	64037.302	5094.437	-86.302
900000	71274.000	65644.796	71362.060	5629.204	-88.060
1000000	78498.000	72382.414	78627.555	6115.586	-129.555
2000000	148933.000	137848.727	149054.844	11084.273	-121.844
3000000	216816.000	201151.622	216970.580	15664.378	-154.580
4000000	283146.000	263126.650	283352.275	20019.350	-206.275
5000000	348513.000	324150.191	348638.140	24362.809	-125.140
6000000	412849.000	384436.227	413076.534	28412.773	-227.534
7000000	476648.000	444122.401	476826.847	32525.599	-178.847
8000000	539777.000	503304.442	539999.729	36472.558	-222.729
9000000	602489.000	562052.636	602676.298	40436.364	-187.298
10000000	664579.000	620420.688	664918.453	44158.312	-339.453

I have computed Li as

$$\begin{aligned} \mathrm{Li}(x) &= 1.045 + \int_2^x \frac{dt}{\log t} \\ &= 1.045 + \int_{\log 2}^{\log x} \frac{e^u du}{u} \text{ (taking } u = \log t \text{ as suggested),} \end{aligned}$$

integrated using the trapezoidal rule with $du = 0.001$.

(The change of variable avoids wasted effort. The original integrand gets progressively flatter as t increases—its derivative with respect to t is $-1/(t(\log t)^2)$—so we ought to make our stepsize progressively larger during the integration. The new integrand becomes progressively steeper, so we have to use a small stepsize for large x; but we are integrating over a much shorter interval!)

Note that the value for Li is not really accurate to 3 digits—it was just convenient to print it that way.

Accuracy of the approximations: For convenience, define $g(x) = x/\log x$.

The Li approximation says that the density of primes *near each* x is $1/\log x$, while the g approximation says that the this is the *average* density on $[1, x]$. Since $1/\log x$ is a decreasing function, this means that $\mathrm{Li}(x) < g(x)$ for all $x > 2$, and that $g(x) - \mathrm{Li}(x)$ increases with x.

From this fact, it seems Li is in some sense a safer approximation than g. $\pi - \mathrm{Li}$ changes sign infinitely often (a remarkable theorem of Littlewood—see Riesel p. 57), and when Li exceeds π, g exceeds it by even more.

However, $\mathrm{Li}(x)/g(x) \to 1$ as $x \to \infty$ (see Riesel p. 46). Eyeballing the table above suggests further that

$$\lim_{x\to\infty} \frac{\pi(x)}{g(x)} = \lim_{x\to\infty} \frac{\pi(x)}{\mathrm{Li}(x)} = 1.$$

Indeed this is known to be the case (the Prime Number Theorem).

Note that neither pi/g nor Li/g approaches 1 monotonically in the above table. This is to be expected. Just compare $pi(n+1)/g(n+1)$ to $pi(n)/g(n)$. If $n + 1$ is prime, then pi increases by 1 while g increases by less than 1; so the ratio increases. If $n + 1$ is not prime, then pi increases by 0 while g increases by more than 0; so the ratio decreases. (The same is true if we replace g by Li.) So on the local scale $(n, n + 1, n + 2, \ldots)$, the quality of the approximations fluctuates up and down. Since the primes are nearly as erratic on the grand scale, we need not expect these fluctuations to be averaged out when we consider $pi(n + 100)/g(n + 100)$ against $pi(n)/g(n)$.

If the Riemann hypothesis is true, then it is known that the absolute error in $\mathrm{Li}(x)$ is (bounded by?) $O(\sqrt{x}\log x)$. (Stated without proof on Riesel p. 60.) This would mean that

$$\mathrm{Li}(x)/\pi(x) = 1 - O((\log x)^2/\sqrt{x}).$$

Riesel mentions some weaker bounds on the absolute error. These other bounds also have order less than $O(\pi(x)) = O(x/\log x)$; otherwise they would not be very interesting!

A closing amusement: I promised at the start to find the first prime beyond a googol. Here it is, using the Miller-Rabin test. My procedure ruled out all the numbers before this one; it failed to rule this one out, with only a 10^{-10} chance of error. (In twice the time, it could have brought the error probability down to 10^{-20}, and so forth.)

```
> (next-prime (expt 10 100))
10000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000267
```

Note that candidates in the range $[10^{100}, 10^{100}+267)$ could not have been ruled out by sieving methods. Though they are all composite, factorizing them would allegedly be NP-hard. (I have verified by the division method that eight of them have no factors below 10^6.)

If you believe the Generalized Riemann Hypothesis, and the correctness of my strong-pseudoprime routine, you can be *sure* that $10^{100} + 267$ is the right answer:

```
> (prime?-GRH (+ 267 (expt 10 100)))
T
```