LMS J. Comput. Math. 15 (2012) 341–359 © 2012 Author
doi:10.1112/S1461157012001088

Efficient implementation of the Hardy–Ramanujan–Rademacher formula

Fredrik Johansson

Abstract

We describe how the Hardy–Ramanujan–Rademacher formula can be implemented to allow the partition function $p(n)$ to be computed with softly optimal complexity $O(n^{1/2+o(1)})$ and very little overhead. A new implementation based on these techniques achieves speedups in excess of a factor 500 over previously published software and has been used by the author to calculate $p(10^{19})$, an exponent twice as large as in previously reported computations. We also investigate performance for multi-evaluation of $p(n)$, where our implementation of the Hardy–Ramanujan–Rademacher formula becomes superior to power series methods on far denser sets of indices than previous implementations. As an application, we determine over 22 billion new congruences for the partition function, extending Weaver's tabulation of 76 065 congruences.

Supplementary materials are available with this article.

1. *Introduction*

Let $p(n)$ denote the number of partitions of n, or the number of ways that n can be written as a sum of positive integers without regard to the order of the terms [**27**, A000041]. The classical way to compute $p(n)$ uses the generating function representation of $p(n)$ combined with Euler's pentagonal number theorem

$$\sum_{n=0}^{\infty} p(n)x^n = \prod_{k=1}^{\infty} \frac{1}{1-x^k} = \left(\sum_{k=-\infty}^{\infty} (-1)^k x^{k(3k-1)/2} \right)^{-1} \tag{1.1}$$

from which one can construct the recursive relation

$$p(n) = \sum_{k=1}^{n} (-1)^{k+1} \left(p\left(n - \frac{k(3k-1)}{2}\right) + p\left(n - \frac{k(3k+1)}{2}\right) \right). \tag{1.2}$$

Equation (1.2) provides a simple and reasonably efficient way to compute the list of values $p(0), p(1), \ldots, p(n-1), p(n)$. Alternatively, applying Fast Fourier Transform (FFT)-based power series inversion to the right-hand side of (1.1) gives an asymptotically faster, essentially optimal algorithm for the same set of values.

An attractive feature of Euler's method, in both the recursive and FFT incarnations, is that the values can be computed more efficiently modulo a small prime number. This is useful for investigating partition function congruences, such as in a recent large-scale computation of $p(n)$ modulo small primes for n up to 10^9 (see [**7**]).

While efficient for computing $p(n)$ for all n up to some limit, Euler's formula is impractical for evaluating $p(n)$ for an isolated, large n. One of the most astonishing number-theoretical discoveries of the 20th century is the Hardy–Ramanujan–Rademacher (HRR) formula, first given as an asymptotic expansion by Hardy and Ramanujan in 1917 [**15**] and subsequently refined to an exact representation by Rademacher in 1936 [**31**], which provides a direct and computationally efficient expression for the single value $p(n)$.

Received 31 October 2011; revised 1 June 2012.

2010 Mathematics Subject Classification 11Y55 (primary), 11-04, 11P83 (secondary).

Supported by Austrian Science Fund (FWF) grant Y464-N18 (Fast Computer Algebra for Special Functions).

Simplified to a first-order estimate, the HRR formula states that

$$p(n) \sim \frac{1}{4n\sqrt{3}}\, e^{\pi\sqrt{2n/3}}, \tag{1.3}$$

from which one gathers that $p(n)$ is a number with roughly $n^{1/2}$ decimal digits. The full version can be stated as

$$p(n) = \sum_{k=1}^{N} \left(\sqrt{\frac{3}{k}}\,\frac{4}{24n-1}\right) A_k(n)\, U\left(\frac{C(n)}{k}\right) + R(n,N), \tag{1.4}$$

$$U(x) = \cosh(x) - \frac{\sinh(x)}{x}, \quad C(n) = \frac{\pi}{6}\sqrt{24n-1}, \tag{1.5}$$

$$A_k(n) = \sum_{h=0}^{k-1} \delta_{\gcd(h,k),1} \exp\left(\pi i \left[s(h,k) - \frac{2hn}{k}\right]\right) \tag{1.6}$$

where $s(h,k)$ is the Dedekind sum

$$s(h,k) = \sum_{i=1}^{k-1} \frac{i}{k}\left(\frac{hi}{k} - \left\lfloor \frac{hi}{k} \right\rfloor - \frac{1}{2}\right) \tag{1.7}$$

and where the remainder satisfies $|R(n,N)| < M(n,N)$ with

$$M(n,N) = \frac{44\pi^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75}\left(\frac{N}{n-1}\right)^{1/2} \sinh\left(\frac{\pi}{N}\sqrt{\frac{2n}{3}}\right). \tag{1.8}$$

It is easily shown that $M(n, cn^{1/2}) \sim n^{-1/4}$ for every positive c. Rademacher's bound (1.8) therefore implies that $O(n^{1/2})$ terms in (1.4) suffice to compute $p(n)$ exactly by forcing $|R(n,N)| < 1/2$ and rounding to the nearest integer. For example, we can take $N = \lceil n^{1/2} \rceil$ when $n \geqslant 65$.

In fact, it was pointed out by Odlyzko [**21**, **26**] that the HRR formula 'gives an algorithm for calculating $p(n)$ that is close to optimal, since the number of bit operations is not much larger than the number of bits of $p(n)$'. In other words, the time complexity should not be much higher than the trivial lower bound $\Omega(n^{1/2})$ derived from (1.3) just for writing down the result. Odlyzko's claim warrants some elaboration, since the HRR formula ostensibly is a triply nested sum containing $O(n^{3/2})$ inner terms.

The computational utility of the HRR formula was, of course, realized long before the availability of electronic computers. For instance, Lehmer [**23**] used it to verify Ramanujan's conjectures $p(599) \equiv 0 \bmod 5^3$ and $p(721) \equiv 0 \bmod 11^2$. Implementations are now available in numerous mathematical software systems, including Pari/GP [**29**], Mathematica [**40**] and Sage [**34**]. However, apart from Odlyzko's remark, we find few algorithmic accounts of the HRR formula in the literature, nor any investigation into the optimality of the available implementations.

The present paper describes a new C implementation of the HRR formula. The code is freely available as a component of the Fast Library for Number Theory (FLINT) [**16**], released under the terms of the GNU General Public License. We show that the complexity for computing $p(n)$ indeed can be bounded by $O(n^{1/2+o(1)})$, and observe that our implementation comes close to being optimal in practice, improving on the speed of previously published software by more than two orders of magnitude.

We benchmark the code by computing some extremely large isolated values of $p(n)$. We also investigate efficiency compared to power series methods for evaluation of multiple values, and finally apply our implementation to the problem of computing congruences for $p(n)$.

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

2. *Simplification of exponential sums*

A naive implementation of formulas (1.4)–(1.7) requires $O(n^{3/2})$ integer operations to evaluate Dedekind sums, and $O(n)$ numerical evaluations of complex exponentials (or cosines, since the imaginary parts ultimately cancel out). In the following section, we describe how the number of integer operations and cosine evaluations can be reduced, for the moment ignoring numerical evaluation.

A first improvement, used for instance in the Sage implementation, is to recognize that Dedekind sums can be evaluated in $O(\log k)$ steps using a GCD-style algorithm, as described by Apostol [**2**], or with Knuth's fraction-free algorithm [**20**] which avoids the overhead of rational arithmetic. This reduces the total number of integer operations to $O(n \log n)$, which is a dramatic improvement but still leaves the cost of computing $p(n)$ quadratic in the size of the final result.

Fortunately, the $A_k(n)$ sums have additional structure as discussed in [**14**, **22**, **24**, **32**, **39**], allowing the computational complexity to be reduced. Since numerous implementers of the HRR formula until now appear to have overlooked these results, it seems appropriate that we reproduce the main formulas and assess the computational issues in more detail. We describe two concrete algorithms: one simple, and one asymptotically fast, the latter being implemented in FLINT.

2.1. *A simple algorithm*

Using properties of the Dedekind eta function, one can derive the formula (which Whiteman [**39**] attributes to Selberg)

$$A_k(n) = \left(\frac{k}{3}\right)^{1/2} \sum_{(3l^2+l)/2 \equiv -n \bmod k} (-1)^l \cos\left(\frac{6l+1}{6k}\pi\right) \tag{2.1}$$

in which the summation ranges over $0 \leqslant l < 2k$ and only $O(k^{1/2})$ terms are nonzero. With a simple brute force search for solutions of the quadratic equation, this representation provides a way to compute $A_k(n)$ that is both simpler and more efficient than the usual definition (1.6).

Although a brute force search requires $O(k)$ loop iterations, the successive quadratic terms can be generated without multiplications or divisions using two coupled linear recurrences. This only costs a few processor cycles per loop iteration, which is a substantial improvement over computing Dedekind sums, and means that the cost up to fairly large k effectively will be dominated by evaluating $O(k^{1/2})$ cosines, adding up to $O(n^{3/4})$ function evaluations for computing $p(n)$.

A basic implementation of (2.1) is given as Algorithm 1. Here the variable m runs over the successive values of $(3l^2 + l)/2$, and r runs over the differences between consecutive m. Various improvements are possible: a modification of the equation allows cutting the loop range in half when k is odd, and the number of cosine evaluations can be reduced by counting the multiplicities of unique angles after reduction to $[0, \pi/4)$, evaluating a weighted sum $\sum w_i \cos(\theta_i)$ at the end, possibly using trigonometric addition theorems to exploit the fact that the differences $\theta_{i+1} - \theta_i$ between successive angles tend to repeat for many different i.

2.2. *A fast algorithm*

From Selberg's formula (2.1), a still more efficient but considerably more complicated multiplicative decomposition of $A_k(n)$ can be obtained. The advantage of this representation is that it only contains $O(\log k)$ cosine factors, bringing the total number of cosine evaluations for $p(n)$ down to $O(n^{1/2} \log n)$. It also reveals exactly when $A_k(n) = 0$ (which is about half the time). We stress that these results are not new; the formulas are given in full detail and with proofs in [**39**].

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

Algorithm 1 Simple algorithm for evaluating $A_k(n)$

Input: Integers $k, n \geqslant 0$
Output: $s = A_k(n)$, where $A_k(n)$ is defined as in (1.6)

```
if k ⩽ 1 then
  return k
else if k = 2 then
  return (−1)^n
end if
(s, r, m) ← (0, 2, (n mod k))
for 0 ⩽ l < 2k do
  if m = 0 then
    s ← s + (−1)^l cos(π(6l + 1)/(6k))
  end if
  m ← m + r
  if m ⩾ k then m ← m − k {m ← m mod k}
  r ← r + 3
  if r ⩾ k then r ← r − k {r ← r mod k}
end for
return (k/3)^{1/2} s
```

First consider the case when k is a power of a prime. Clearly $A_1(n) = 1$ and $A_2(n) = (-1)^n$. Otherwise let $k = p^\lambda$ and $v = 1 - 24n$. Then, using the notation $(a|m)$ for Jacobi symbols to avoid confusion with fractions, we have

$$A_k(n) = \begin{cases} (-1)^\lambda (-1|m_2) k^{1/2} \sin(4\pi m_2/8k) & \text{if } p = 2 \\ 2(-1)^{\lambda+1} (m_3|3)(k/3)^{1/2} \sin(4\pi m_3/3k) & \text{if } p = 3 \\ 2(3|k) k^{1/2} \cos(4\pi m_p/k) & \text{if } p > 3 \end{cases} \tag{2.2}$$

where m_2, m_3 and m_p respectively are any solutions of

$$(3m_2)^2 \equiv v \mod 8k \tag{2.3}$$

$$(8m_3)^2 \equiv v \mod 3k \tag{2.4}$$

$$(24m_p)^2 \equiv v \mod k \tag{2.5}$$

provided, when $p > 3$, that such an m_p exists and that $\gcd(v, k) = 1$. If, on the other hand, $p > 3$ and either of these two conditions do not hold, we have

$$A_k(n) = \begin{cases} 0 & \text{if } v \text{ is not a quadratic residue modulo } k \\ (3|k) k^{1/2} & \text{if } v \equiv 0 \bmod p, \lambda = 0 \\ 0 & \text{if } v \equiv 0 \bmod p, \lambda > 1. \end{cases} \tag{2.6}$$

If k is not a prime power, assume that $k = k_1 k_2$ where $\gcd(k_1, k_2) = 1$. Then we can factor $A_k(n)$ as $A_k(n) = A_{k_1}(n_1) A_{k_2}(n_2)$, where n_1, n_2 are any solutions of the following equations. If $k_1 = 2$, then

$$\begin{cases} 32n_2 \equiv 8n + 1 \mod k_2 \\ n_1 \equiv n - (k_2^2 - 1)/8 \mod 2, \end{cases} \tag{2.7}$$

if $k_1 = 4$, then

$$\begin{cases} 128n_2 \equiv 8n + 5 \mod k_2 \\ k_2^2 n_1 \equiv n - 2 - (k_2^2 - 1)/8 \mod 4, \end{cases} \tag{2.8}$$

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

and if k_1 is odd or divisible by 8, then

$$\begin{cases} k_2^2 d_2 e n_1 \equiv d_2 e n + (k_2^2 - 1)/d_1 & \mod k_1 \\ k_1^2 d_1 e n_2 \equiv d_1 e n + (k_1^2 - 1)/d_2 & \mod k_2 \end{cases} \tag{2.9}$$

where $d_1 = \gcd(24, k_1)$, $d_2 = \gcd(24, k_2)$, $24 = d_1 d_2 e$.

Here $(k^2 - 1)/d$ denotes an operation done on integers, rather than a modular division. All other solving steps in (2.2)–(2.9) amount to computing greatest common divisors, carrying out modular ring operations, finding modular inverses, and computing modular square roots. Repeated application of these formulas results in Algorithm 2, where we omit the detailed arithmetic for brevity.

Algorithm 2 Fast algorithm for evaluating $A_k(n)$

Input: Integers $k \geqslant 1$, $n \geqslant 0$
Output: $s = A_k(n)$, where $A_k(n)$ is defined as in (1.6)

```
Compute the prime factorization k = p_1^{λ_1} p_2^{λ_2} ... p_j^{λ_j}
s ← 1
for 1 ⩽ i ⩽ j and while s ≠ 0 do
  if i < j then
    (k_1, k_2) ← (p_i^{λ_i}, k/p_i^{λ_i})
    Compute n_1, n_2 by solving the respective case of (2.7)–(2.9)
    s ← s × A_{k_1}(n_1) {Handle the prime power case using (2.2)–(2.6)}
    (k, n) ← (k_2, n_2)
  else
    s ← s × A_k(n) {Prime power case}
  end if
end for
return s
```

2.3. *Computational cost*

A precise complexity analysis of Algorithm 2 should take into account the cost of integer arithmetic. Multiplication, division, computation of modular inverses, greatest common divisors and Jacobi symbols of integers bounded in absolute value by $O(k)$ can all be performed with bit complexity $O(\log^{1+o(1)} k)$.

At first sight, integer factorization might seem to pose a problem. We can, however, factor all indices k summed over in (1.4) in $O(n^{1/2} \log^{1+o(1)} n)$ bit operations. For example, using the sieve of Eratosthenes, we can precompute a list of length $n^{1/2}$ where entry k is the largest prime dividing k.

A fixed index k is a product of at most $O(\log k)$ prime powers with exponents bounded by $O(\log k)$. For each prime power, we need $O(1)$ operations with roughly the cost of multiplication, and $O(1)$ square roots, which are the most expensive operations.

To compute square roots modulo p^λ, we can use the Tonelli–Shanks algorithm [**33, 36**] or Cipolla's algorithm [**8**] modulo p followed by Hensel lifting up to p^λ. Assuming that we know a quadratic nonresidue modulo p, the Tonelli–Shanks algorithm requires $O(\log^3 k)$ multiplications in the worst case and $O(\log^2 k)$ multiplications on average, while Cipolla's algorithm requires $O(\log^2 k)$ multiplications in the worst case [**9**]. This puts the bit complexity of factoring a single exponential sum $A_k(n)$ at $O(\log^{3+o(1)} k)$, and gives us the following result.

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

THEOREM 1. *Assume that we know a quadratic nonresidue modulo p for all primes p up to $n^{1/2}$. Then we can factor all the $A_k(n)$ required for evaluating $p(n)$ using $O(n^{1/2}\log^{3+o(1)} n)$ bit operations.*

The assumption in Theorem 1 can be satisfied with a precomputation that does not affect the complexity. If $n_2(p_k)$ denotes the least quadratic nonresidue modulo the kth prime number, it is a theorem of Erdős [**10, 30**] that as $x \to \infty$,

$$\frac{1}{\pi(x)} \sum_{p_k \leqslant x} n_2(p_k) \to \sum_{k=1}^{\infty} \frac{p_k}{2^k} = C < 3.675. \tag{2.10}$$

Given the primes up to $x = n^{1/2}$, we can therefore build a table of nonresidues by testing no more than $(C + o(1))\pi(n^{1/2})$ candidates. Since $\pi(n^{1/2}) = O(n^{1/2}/\log n)$ and a quadratic residue test takes $O(\log^{1+o(1)} p)$ time, the total precomputation time is $O(n^{1/2}\log^{o(1)} n)$.

In practice, it is sufficient to generate nonresidues on the fly since $O(1)$ candidates need to be tested on average, but we can only prove an $O(\log^c k)$ bound for factoring an isolated $A_k(n)$ by assuming the Extended Riemann Hypothesis which gives $n_2(p) = O(\log^2 p)$ [**1**].

2.4. *Implementation notes*

As a matter of practical efficiency, the modular arithmetic should be done with as little overhead as possible. FLINT provides optimized routines for arithmetic with moduli smaller than 32 or 64 bits (depending on the hardware word size) which are used throughout; including, among other things, a binary-style GCD algorithm, division and remainder using precomputed inverses, and supplementary code for operations on two-limb (64 or 128 bit) integers.

We note that since $A_k(n) = A_k(n + k)$, we can always reduce n modulo k, and perform all modular arithmetic with moduli up to some small multiple of k. In principle, the implementation of the modular arithmetic in FLINT thus allows calculating $p(n)$ up to approximately $n = (2^{64})^2 \approx 10^{38}$ on a 64-bit system, which roughly equals the limit on n imposed by the availability of addressable memory to store $p(n)$.

At present, our implementation of Algorithm 2 simply calls the FLINT routine for integer factorization repeatedly rather than sieving over the indices. Although convenient, this technically results in a higher total complexity than $O(n^{1/2+o(1)})$. However, the code for factoring single-word integers, which uses various optimizations for small factors and Hart's 'One Line Factor' variant of Lehman's method to find large factors [**17**], is fast enough that integer factorization only accounts for a small fraction of the running time for any feasible n. If needed, full sieving could easily be added in the future.

Likewise, the square root function in FLINT uses the Tonelli–Shanks algorithm and generates a nonresidue modulo p on each call. This is suboptimal in theory but efficient enough in practice.

3. *Numerical evaluation*

We now turn to the problem of numerically evaluating (1.4)–(1.5) using arbitrary-precision arithmetic, given access to Algorithm 2 for symbolically decomposing the $A_k(n)$ sums. Although (1.8) bounds the truncation error in the HRR series, we must also account for the effects of having to work with finite-precision approximations of the terms.

3.1. *Floating-point precision*

We assume the use of variable-precision binary floating-point arithmetic (a simpler but less efficient alternative, avoiding the need for detailed manual error bounds, would be to use arbitrary-precision interval arithmetic). Basic notions about floating-point arithmetic and error analysis can be found in [**18**].

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

If the precision is r bits, we let $\varepsilon = 2^{-r}$ denote the unit roundoff. We use the symbol $\hat{x}$ to signify a floating-point approximation of an exact quantity x, having some relative error $\delta = (\hat{x} - x)/x$ when $x \neq 0$. If $\hat{x}$ is obtained by rounding x to the nearest representable floating-point number (at most 0.5 ulp error) at precision r, we have $|\delta| \leqslant \varepsilon$. Except where otherwise noted, we assume correct rounding to nearest.

A simple strategy for computing $p(n)$ is as follows. For a given n, we first determine an N such that $|R(n, N)| < 0.25$, for example using a linear search. A tight upper bound for $\log_2 M(n, N)$ can be computed easily using low-precision arithmetic. We then approximate the kth term t_k using a working precision high enough to guarantee

$$|\hat{t}_k - t_k| \leqslant \frac{0.125}{N}, \tag{3.1}$$

and perform the outer summation such that the absolute error of each addition is bounded by $0.125/N$. This clearly guarantees $|\hat{p}(n) - p(n)| < 0.5$, allowing us to determine the correct value of $p(n)$ by rounding to the nearest integer. We might, alternatively, carry out the additions exactly and save one bit of precision for the terms.

In what follows, we derive a simple but essentially asymptotically tight expression for a working precision, varying with k, sufficiently high for (3.1) to hold. Using Algorithm 2, we write the term to be evaluated in terms of exact integer parameters $\alpha, \beta, a, b, p_i, q_i$ as

$$t_k = \frac{\alpha}{\beta} \frac{\sqrt{a}}{\sqrt{b}} U\left(\frac{C}{k}\right) \prod_{i=1}^{m} \cos\left(\frac{p_i \pi}{q_i}\right). \tag{3.2}$$

LEMMA 2. *Let $p \in \mathbb{Z}$, $q \in \mathbb{N}^+$ and let r be a precision in bits with $2^r > \max(3q, 64)$. Suppose that* sin *and* cos *can be evaluated on $(0, \pi/4)$ with relative error at most 2ε for floating-point input, and suppose that π can be approximated with relative error at most ε. Then we can evaluate $\cos(p\pi/q)$ with relative error less than 5.5ε.*

Proof. We first reduce p and q with exact integer operations so that $0 < 4p < q$, giving an angle in the interval $(0, \pi/4)$. Then we approximate $x = p\pi/q$ using three roundings, giving $\hat{x} = x(1 + \delta_x)$ where $|\delta_x| \leqslant (1 + \varepsilon)^3 - 1$. The assumption $\varepsilon < 1/(3q)$ gives $(q/(q-1))(1 + \delta_x) < 1$ and therefore also $\hat{x} \in (0, \pi/4)$.

Next, we evaluate $f(\hat{x})$ where $f = \pm\cos$ or $f = \pm\sin$ depending on the argument reduction. By Taylor's theorem, we have $f(\hat{x}) = f(x)(1 + \delta'_x)$ where

$$|\delta'_x| = \frac{|f(\hat{x}) - f(x)|}{f(x)} = \frac{x|\delta_x||f'(\xi)|}{f(x)} \tag{3.3}$$

for some ξ between x and $\hat{x}$, giving $|\delta'_x| \leqslant (\frac{1}{4}\pi\sqrt{2})|\delta_x|$. Finally, rounding results in

$$\hat{f}(\hat{x}) = f(x)(1 + \delta) = f(x)(1 + \delta'_x)(1 + \delta_f)$$

where $|\delta_f| \leqslant 2\varepsilon$. The inequality $\varepsilon < 1/64$ gives $|\delta| < 5.5\varepsilon$. □

To obtain a simple error bound for $U(x)$ where $x = C/k$, we make the somewhat crude restriction that $n > 2000$. We also assume $k < n^{1/2}$ and $x > 3$, which are not restrictions: if N is chosen optimally using Rademacher's remainder bound (1.8), the maximum k decreases and the minimum x increases with larger n. In particular, $n > 2000$ is sufficient with Rademacher's bound (or any tighter bound for the remainder).

We assume that C is precomputed; of course, this only needs to be done once during the calculation of $p(n)$, at a precision a few bits higher than that of the $k = 1$ term.

LEMMA 3. *Suppose $n > 2000$ and let r be a precision in bits such that $2^r > \max(16n^{1/2}, 2^{10})$. Let $x = C/k$ where C is defined as in (1.5) and where k is constrained such that $k < n^{1/2}$*

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

and $x > 3$. Assume that $\hat{C} = C(n)(1+\delta_C)$ has been precomputed with $|\delta_C| \leqslant 2\varepsilon$ and that sinh *and* cosh *can be evaluated with relative error at most 2ε for floating-point input. Then we can evaluate $U(x)$ with relative error at most $(9x+15)\varepsilon$.*

Proof. We first compute $\hat{x} = x(1+\delta_x) = (C/k)(1+\delta_C)(1+\delta_0)$ where $|\delta_0| \leqslant \varepsilon$. Next, we compute

$$\hat{U}(\hat{x}) = U(\hat{x})(1+\delta_U) = U(x)(1+\delta'_x)(1+\delta_U) = U(x)(1+\delta) \tag{3.4}$$

where we have to bound the error δ'_x propagated in the composition as well as the rounding error δ_U in the evaluation of $U(\hat{x})$. Using the inequality $x|\delta_x| < 4x\varepsilon < \log 2$, we have

$$|\delta'_x| \leqslant \frac{x|\delta_x|U'(x+x|\delta_x|)}{U(x)} \leqslant \frac{x|\delta_x|\exp(x+x|\delta_x|)}{2U(x)} \leqslant \frac{x|\delta_x|\exp(x)}{U(x)} \leqslant 3x|\delta_x|. \tag{3.5}$$

Evaluating $U(\hat{x})$ using the obvious sequence of operations results in

$$|\delta_U| = \frac{\left|\left(\cosh(\hat{x})(1+2\delta_1) - \dfrac{\sinh(\hat{x})}{\hat{x}}(1+2\delta_2)(1+\delta_3)\right)(1+\delta_4) - U(\hat{x})\right|}{U(\hat{x})} \tag{3.6}$$

where $|\delta_i| \leqslant \varepsilon$ and $\hat{x} > z$ where $z = 3(1-4\varepsilon)$. This expression is maximized by setting $\hat{x}$ as small as possible and taking $\delta_1 = \delta_4 = -\delta_2 = -\delta_3 = \varepsilon$, which gives

$$|\delta_U| < \frac{\cosh(z)}{U(z)}\varepsilon(3+2\varepsilon) + \frac{\sinh(z)}{z\,U(z)}\varepsilon(2+\varepsilon-2\varepsilon^2) < 5.5\varepsilon. \tag{3.7}$$

Expanding (3.4) using (3.5) and (3.7) gives $|\delta| < \varepsilon(5.5 + 9x + 56x\varepsilon + 33x\varepsilon^2)$. Finally, we obtain $5.5 + 56x\varepsilon + 33x\varepsilon^2 < 15$ by a direct application of the assumptions. □

Put together, assuming floating-point implementations of standard transcendental functions with at most 1 ulp error (implying a relative error of at most 2ε), correctly rounded arithmetic and the constant π, we have the following theorem.

THEOREM 4. *Let $n > 2000$. For (3.1) to hold, it is sufficient to evaluate (3.2) using a precision of $r = \max(\log_2 N + \log_2 |t_k| + \log_2(10x+7m+22) + 3,\ \frac{1}{2}\log_2 n + 5,\ 11)$ bits.*

Proof. We can satisfy the assumptions of Lemmas 2 and 3. In particular, $3q \leqslant 24k < 24n^{1/2} < 2^r$. The top-level arithmetic operations in (3.2), including the square roots, amount to a maximum of $m+6$ roundings. Lemmas 2 and 3 and elementary inequalities give the relative error bound

$$|\delta| < (1+\varepsilon)^{m+6}(1+5.5\varepsilon)^m(1+(15+9x)\varepsilon) - 1 \tag{3.8}$$

$$< \left(1 + \frac{(m+6)\varepsilon}{1-(m+6)\varepsilon}\right)\left(1 + \frac{5.5m\varepsilon}{1-5.5m\varepsilon}\right)(1+(15+9x)\varepsilon) - 1 \tag{3.9}$$

$$= \frac{21\varepsilon + 6.5m\varepsilon - 33m\varepsilon^2 - 5.5m^2\varepsilon^2 + 9x\varepsilon}{(1-5.5\varepsilon m)(1-\varepsilon(m+6))} \tag{3.10}$$

$$< (10x+7m+22)\varepsilon. \tag{3.11}$$

The result follows by taking logarithms in (3.1). □

To make Theorem 4 effective, we can use $m \leqslant \log_2 k$ and bound $|t_k|$ using (1.4) with $|A_k| \leqslant k$ and $U(x) < e^x/2$, giving

$$\log|t_k| < \frac{(24n-1)^{1/2}\,\pi}{6k} + \frac{\log k}{2} - \log(24n-1) + \left(\log 2 + \frac{\log 3}{2}\right). \tag{3.12}$$

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

Naturally, for $n \leqslant 2000$, the same precision bound can be verified to be sufficient through direct computation. We can even reduce overhead for small n by using a tighter precision, say $r = |t_k| + O(1)$, up to some limit small enough to be tested exhaustively (perhaps much larger than 2000). The requirement that $r > \frac{1}{2}\log_2 n + O(1)$ always holds in practice if we set a minimum precision; for n feasible on present hardware, it is sufficient to never drop below IEEE double (53-bit) precision.

3.2. *Computational cost*

We assume that r-bit floating-point numbers can be multiplied in time $M(r) = O(r \log^{1+o(1)} r)$. It is well known (see [5]) that the elementary functions exp, log, sin etc. can be evaluated in time $O(M(r) \log r)$ using methods based on the arithmetic–geometric mean (AGM). A popular alternative is binary splitting, which typically has cost $O(M(r) \log^2 r)$ but tends to be faster than the AGM in practice.

To evaluate $p(n)$ using the HRR formula, we must add $O(n^{1/2})$ terms each of which can be written as a product of $O(\log k)$ factors. According to (3.12) and the error analysis in the previous section, the kth term needs to be evaluated to a precision of $O(n^{1/2}/k) + O(\log n)$ bits. Using any combination of $O(M(r) \log^{\alpha} r)$ algorithms for elementary functions, the complexity of the numerical operations is

$$O\left(\sum_{k=1}^{n^{1/2}} \log k \; M\left(\frac{n^{1/2}}{k}\right) \log^{\alpha} \frac{n^{1/2}}{k}\right) = O(n^{1/2} \log^{\alpha+3+o(1)} n) \tag{3.13}$$

which is nearly optimal in the size of the output. Combined with the cost of the factoring stage, the complexity for the computation of $p(n)$ as a whole is therefore, when properly implemented, softly optimal at $O(n^{1/2+o(1)})$. From (3.13) with the best known complexity bound for elementary functions, we obtain the following.

THEOREM 5. *The value $p(n)$ can be computed in time $O(n^{1/2} \log^{4+o(1)} n)$.*

A subtle but crucial detail in this analysis is that the additions in the main sum must be implemented in such a way that they have cost $O(n^{1/2}/k)$ rather than $O(n^{1/2})$, since the latter would result in an $O(n)$ total complexity. If the additions are performed in-place in memory, we can perform summations the natural way and rely on carry propagation terminating in an expected $O(1)$ steps, but many implementations of arbitrary-precision floating-point arithmetic do not provide this optimization.

One way to solve this problem is to add the terms in reverse order, using a precision that matches the magnitude of the partial sums. Or, if we add the terms in forward order, we can amortize the cost by keeping separate summation variables for the partial sums of terms not exceeding $r_1, r_1/2, r_1/4, r_1/8, \ldots$ bits.

3.3. *Arithmetic implementation*

FLINT uses the MPIR library, derived from GMP, for arbitrary-precision arithmetic, and the MPFR library on top of MPIR for asymptotically fast arbitrary-precision floating-point numbers and correctly rounded transcendental functions [**11**, **13**, **25**]. Thanks to the strong correctness guarantees of MPFR, it is relatively straightforward to write a provably correct implementation of the partition function using Theorem 4.

Although the default functions provided by MPFR are quite fast, order-of-magnitude speedups were found possible with custom routines for parts of the numerical evaluation. An unfortunate consequence is that our implementation currently relies on routines that, although heuristically sound, have not yet been proved correct, and perhaps are more likely to contain implementation bugs than the well-tested standard functions in MPFR.

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

All such heuristic parts of the code are, however, well isolated, and we expect that they can be replaced with rigorous versions with equivalent or better performance in the future.

3.4. *Hardware arithmetic*

Inspired by the Sage implementation, which was written by Jonathan Bober, our implementation switches to hardware (IEEE double) floating-point arithmetic to evaluate (3.2) when the precision bound falls below 53 bits. This speeds up evaluation of the 'long tail' of terms with very small magnitude.

Using hardware arithmetic entails some risk. Although the IEEE floating-point standard implemented on all modern hardware guarantees 0.5 ulp error for arithmetic operations, accuracy may be lost, for example, if the compiler generates long-double instructions which trigger double rounding, or if the rounding mode of the processor has been changed.

We need to be particularly concerned about the accuracy of transcendental functions. The hardware transcendental functions on the Intel Pentium processor and its descendants guarantee an error of at most 1 ulp when rounding to nearest [**19**], as do the software routines in the portable and widely used FDLIBM library [**35**]. Nevertheless, some systems may be equipped with poorer implementations.

Fortunately, the bound (1.8) and Theorem 4 are lax enough in practice that errors up to a few ulp can be tolerated, and we expect any reasonably implemented double-precision transcendental functions to be adequate. Most importantly, range reducing the arguments of trigonometric functions to $(0, \pi/4)$ avoids catastrophic error for large arguments which is a misfeature of some implementations.

3.5. *High-precision evaluation of exponentials*

MPFR implements the exponential and hyperbolic functions using binary splitting at high precision, which is asymptotically fast up to logarithmic factors. We can, however, improve performance by not computing the hyperbolic functions in $U(x)$ from scratch when k is small. Instead, we precompute $\exp(C)$ with the initial precision of C, and then compute $(\cosh(C/k), \sinh(C/k))$ from $(\exp(C))^{1/k}$; that is, by kth root extractions which have cost $O((\log k)M(r))$. Using the builtin MPFR functions, root extraction was found experimentally to be faster than evaluating the exponential function up to approximately $k = 35$ over a large range of precisions.

For extremely large n, we also speed up computation of the constant C by using binary splitting to compute π (adapting code written by H. Xue [**12**]) instead of the default function in MPFR, which uses arithmetic–geometric mean iteration. As has been pointed out previously [**41**], binary splitting is more than four times faster for computing π in practice, despite theoretically having a log factor worse complexity. When evaluating $p(n)$ for multiple values of n, the value of π should of course be cached, which MPFR does automatically.

3.6. *High-precision cosines*

The MPFR cosine and sine functions implement binary splitting, with similar asymptotics as the exponential function. At high precision, our implementation switches to custom code for evaluating $\alpha = \cos(p\pi/q)$ when q is not too large, taking advantage of the fact that α is an algebraic number. Our strategy consists of generating a polynomial P such that $P(\alpha) = 0$ and solving this equation using Newton iteration, starting from a double precision approximation of the desired root. Using a precision that doubles with each step of the Newton iteration, the complexity is $O(\deg(P)M(r))$.

The numbers $\cos(p\pi/q)$ are computed from scratch as needed: caching values with small p and q was found to provide a negligible speedup while needlessly increasing memory consumption and code complexity.

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

Our implementation uses the minimal polynomial $\Phi_n(x)$ of $\cos(2\pi/n)$, which has degree $d = \phi(n)/2$ for $n \geqslant 3$ [37]. More precisely, we use the scaled polynomial $2^d\Phi(x) \in \mathbb{Z}[x]$. This polynomial is looked up from a precomputed table when n is small, and otherwise is generated using a balanced product tree, starting from floating-point approximations of the conjugate roots. As a side remark, this turned out to be around a thousand times faster than computing the minimal polynomial with the standard commands in either Sage or Mathematica.

We sketch the procedure for high-precision evaluation of $\cos(p\pi/q)$ as Algorithm 3, omitting various special cases and implementation details (for example, our implementation performs the polynomial multiplications over $\mathbb{Z}[x]$ by embedding the approximate coefficients as fixed-point numbers).

Algorithm 3 High-precision numerical evaluation of $\cos(p\pi/q)$

Input: Coprime integers p and q with $q \geqslant 3$, and a precision r

Output: An approximation of $\cos(p\pi/q)$ accurate to r bits

$n \leftarrow (1 + (p \bmod 2))\, q$

$d \leftarrow \phi(n)/2$

{Bound coefficients in $2^d \prod_{i=1}^{d}(x - \alpha)$}

$b \leftarrow \lceil \log_2 d \rceil + \left\lceil \log_2 \binom{d}{d/2} \right\rceil$

{Use a balanced product tree and a precision of $b + O(\log d)$ bits}

$\Phi \leftarrow 2^d \prod_{i=1,\gcd(i,n)=1}^{\deg(\Phi)\leqslant d}(x - \cos(i\pi/n))$ {Use base case algorithm for cos}

{Round to an integer polynomial}

$\Phi \leftarrow \sum_{k=0}^{d} \left\lfloor [x^k]\Phi + \frac{1}{2} \right\rfloor x^k$

Compute precisions $r_0 = r + 8, r_1 = r_0/2 + 8, \ldots, r_j = r_{j-1}/2 + 8 < 50$

$x \leftarrow \cos(p\pi/q)$ {To 50 bits, using base case algorithm}

for $i \leftarrow j - 1, j - 2 \ldots 0$ **do**

  {Evaluate using the Horner scheme at $r_i + b$ bit precision}

  $x \leftarrow x - \Phi(x)/\Phi'(x)$

end for

return x

We do not attempt to prove that the internal precision management of Algorithm 3 is correct. However, the polynomial generation can easily be tested up to an allowed bound for q, and the function can be tested to be correct for all pairs p, q at some fixed, high precision r. We may then argue heuristically that the well-behavedness of the object function in the root-finding stage combined with the highly conservative padding of the precision by several bits per iteration suffices to ensure full accuracy at the end of each step in the final loop, given an arbitrary r.

A different way to generate $\Phi_n(x)$ using Chebyshev polynomials is described in [37]. One can also use the squarefree part of an offset Chebyshev polynomial

$$P(x) = \frac{T_{2q}(x) - 1}{\gcd(T_{2q}(x) - 1, T'_{2q}(x))}$$

directly, although this is somewhat less efficient than the minimal polynomial.

Alternatively, since $\cos(p\pi/q)$ is the real part of a root of unity, the polynomial $x^q - 1$ could be used. The use of complex arithmetic adds overhead, but the method would be faster for large q since x^q can be computed in time $O((\log q)M(r))$ using repeated squaring. We also note that the secant method could be used instead of the standard Newton iteration in Algorithm 3. This increases the number of iterations, but removes the derivative evaluation, possibly providing some speedup.

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

In our implementation, Algorithm 3 was found to be faster than the MPFR trigonometric functions for $q < 250$ roughly when the precision exceeds $400 + 4q^2$ bits. This estimate includes the cost of generating the minimal polynomial on the fly.

3.7. *The main algorithm*

Algorithm 4 outlines the main routine in FLINT with only minor simplifications. To avoid possible corner cases in the convergence of the HRR sum, and to avoid unnecessary overhead, values with $n < 128$ (exactly corresponding to $p(n) < 2^{32}$) are looked up from a table. We only use k, n, N in Theorem 4 in order to make the precision decrease uniformly, allowing amortized summation to be implemented in a simple way.

Algorithm 4 Main routine implementing the HRR formula

Input: $n \geqslant 128$
Output: $p(n)$
  Determine N and initial precision r_1 using Theorem 4
  $C \leftarrow \frac{\pi}{6}\sqrt{24n-1}$ {At $r_1 + 3$ bits}
  $u \leftarrow \exp(C)$
  $s_1 \leftarrow s_2 \leftarrow 0$
  for $1 \leqslant k \leqslant N$ **do**
    Write term k as (3.2) by calling Algorithm 2
    if $A_k(n) \neq 0$ **then**
      Determine term precision r_k for $|t_k|$ using Theorem 4
      {Use Algorithm 3 if $q_i < 250$ and $r_k > 400 + 4q^2$}
      $t \leftarrow (-1)^s \sqrt{a/b} \prod \cos(p_i \pi / q_i)$
      $t \leftarrow t \times U(C/k)$ {Compute U from $u^{1/k}$ if $k < 35$}
      {Amortized summation: $r(s_2)$ denotes precision of the variable s_2}
      $s_2 \leftarrow s_2 + t$
      if $2r_k < r(s_2)$ **then**
        $s_1 \leftarrow s_1 + s_2$ {Exactly or with precision exceeding r_1}
        $r(s_2) \leftarrow r_k$ {Change precision}
        $s_2 \leftarrow 0$
      end if
    end if
  end for
  return $\lfloor s_1 + s_2 + \frac{1}{2} \rfloor$

Since our implementation presently relies on some numerical heuristics (and in any case, considering the intricacy of the algorithm), care has been taken to test it extensively. All $n \leqslant 10^6$ have been checked explicitly, and a large number of isolated $n \gg 10^6$ have been compared against known congruences and values computed with Sage and Mathematica.

As a strong robustness check, we observe experimentally that the numerical error in the final sum decreases with larger n. For example, the error is consistently smaller than 10^{-3} for $n > 10^6$ and smaller than 10^{-4} for $n > 10^9$. This phenomenon reflects the fact that (1.8) overshoots the actual magnitude of the terms with large k, combined with the fact that rounding errors average out pseudorandomly rather than approaching worst-case bounds.

4. *Benchmarks*

Table 1 and Figure 1 compare performance of Mathematica 7, Sage 4.7 and FLINT on a laptop with a Pentium T4400 2.2 GHz CPU and 3 GB of RAM, running 64 bit Linux. To the author's

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

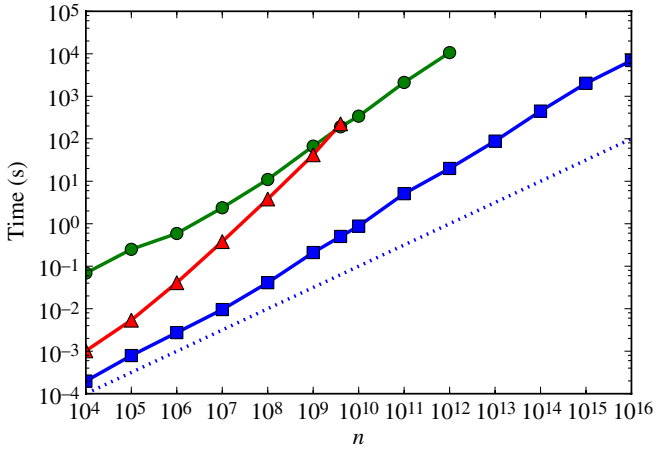

FIGURE 1. *CPU time t in seconds for computing $p(n)$: FLINT (blue squares), Mathematica 7 (green circles), Sage 4.7 (red triangles). The dotted line shows $t = 10^{-6}n^{1/2}$, indicating the slope of an idealized algorithm satisfying the trivial lower complexity bound $\Omega(n^{1/2})$ (the offset 10^{-6} is arbitrary).*

knowledge, Mathematica and Sage contain the fastest previously available partition function implementations by far.

The FLINT code was run with MPIR version 2.4.0 and MPFR version 3.0.1. Since Sage 4.7 uses an older version of MPIR and Mathematica is based on an older version of GMP, differences in performance of the underlying arithmetic slightly skew the comparison, but probably not by more than a factor of two.

The limited memory of the aforementioned laptop restricted the range of feasible n to approximately 10^{16}. Using a system with an AMD Opteron 6174 processor and 256 GB RAM allowed calculating $p(10^{17})$, $p(10^{18})$ and $p(10^{19})$ as well. The last computation took just less than 100 hours and used more than 150 GB of memory, producing a result with over 11 billion bits. Some large values of $p(n)$ are listed in Table 2.

TABLE 1. *Timings for computing $p(n)$ in Mathematica 7, Sage 4.7 and FLINT up to $n = 10^{16}$ on the same system, as well as FLINT timings for $n = 10^{17}$–10^{19} (*) done on different (slightly faster) hardware. Calculations running less than one second were repeated, allowing benefits from data caching. The rightmost column shows the amount of time in the FLINT implementation spent computing the first term.*

n	Mathematica 7	Sage 4.7	FLINT	Initial (%)
10^4	69 ms	1 ms	0.20 ms	
10^5	250 ms	5.4 ms	0.80 ms	
10^6	590 ms	41 ms	2.74 ms	
10^7	2.4 s	0.38 s	0.010 s	
10^8	11 s	3.8 s	0.041 s	
10^9	67 s	42 s	0.21 s	43
10^{10}	340 s		0.88 s	53
10^{11}	2 116 s		5.1 s	48
10^{12}	10 660 s		20 s	49
10^{13}			88 s	48
10^{14}			448 s	47
10^{15}			2 024 s	39
10^{16}			6 941 s	45
10^{17}			27 196* s	33
10^{18}			87 223* s	38
10^{19}			350 172* s	39

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

As can be seen in Table 1 and Figure 1, the FLINT implementation exhibits a time complexity only slightly higher than $O(n^{1/2})$, with a comparatively small constant factor. The Sage implementation is fairly efficient for small n but has a complexity closer to $O(n)$, and is limited to arguments $n < 2^{32} \approx 4 \times 10^9$.

The Mathematica implementation appears to have complexity slightly higher than $O(n^{1/2})$ as well, but consistently runs about 200–500 times slower than our implementation. Based on extrapolation, computing $p(10^{19})$ would take several years. It is unclear whether Mathematica is actually using a nearly-optimal algorithm or whether the slow growth is just the manifestation of various overheads dwarfing the true asymptotic behavior. The ratio compared to FLINT appears too large to be explained by differences in performance of the underlying arithmetic alone; for example, evaluating the first term in the series for $p(10^{10})$ to required precision in Mathematica only takes about one second.

We get one external benchmark from [4], where it is reported that Crandall computed $p(10^9)$ in three seconds on a laptop in December 2008, 'using the HRR 'finite' series for $p(n)$ along with FFT methods'. Even accounting for possible hardware differences, this appears to be an order of magnitude slower than our implementation.

4.1. *Optimality relative to the first term*

Table 1 includes time percentages spent on evaluating the first term, $\exp(C)$, in the FLINT implementation. We find that this step fairly consistently amounts to just a little less than half of the running time. Our implementation is therefore nearly optimal in a practical sense, since the first term in the HRR expansion hardly can be avoided and at most a factor of two can be gained by improving the tail evaluation.

Naturally, there is some potential to implement a faster version of the exponential function than the one provided by MPFR, reducing the cost of the first term. Improvements on the level of bignum multiplication would, on the other hand, presumably have a comparatively uniform effect.

By similar reasoning, at most a factor of two can be gained through parallelization of our implementation by assigning terms in the HRR sum to separate threads. Further speedup on a multicore system requires parallelized versions of lower level routines, such as the exponential function or bignum multiplication. (A simplistic multithreaded version of the FLINT partition function was tested for n up to 10^{18}, giving nearly a twofold speedup on two cores, but failed when computing 10^{19} for reasons yet to be determined.) Fortunately, it is likely to be more interesting in practice to be able to evaluate $p(n)$ for a range of large values than just for a single value, and this task naturally parallelizes well.

5. *Multi-evaluation and congruence generation*

One of the central problems concerning the partition function is the distribution of values of $p(n) \bmod m$. In 2000, Ono [28] proved that for every prime $m \geqslant 5$, there exist infinitely many

TABLE 2. *Large values of $p(n)$. The table also lists the number of terms N in the HRR formula used by FLINT (theoretically bounding the error by 0.25) and the difference between the floating-point sum and the rounded integer.*

n	Decimal expansion	Number of digits	Terms	Error
10^{12}	6129000962 . . . 6867626906	1 113 996	264 526	2×10^{-7}
10^{13}	5714414687 . . . 4630811575	3 522 791	787 010	3×10^{-8}
10^{14}	2750960597 . . . 5564896497	11 140 072	2 350 465	-1×10^{-8}
10^{15}	1365537729 . . . 3764670692	35 228 031	7 043 140	-3×10^{-9}
10^{16}	9129131390 . . . 3100706231	111 400 846	21 166 305	-9×10^{-10}
10^{17}	8291300791 . . . 3197824756	352 280 442	63 775 038	5×10^{-10}
10^{18}	1478700310 . . . 1701612189	1 114 008 610	192 605 341	4×10^{-10}
10^{19}	5646928403 . . . 3674631046	3 522 804 578	582 909 398	4×10^{-11}

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

congruences of the type

$$p(Ak + B) \equiv 0 \mod m \tag{5.1}$$

where A, B are fixed and k ranges over all nonnegative integers. Ono's proof is nonconstructive, but Weaver [38] subsequently gave an algorithm for finding congruences of this type when $m \in \{13, 17, 19, 23, 29, 31\}$, and used the algorithm to compute 76 065 explicit congruences.

Weaver's congruences are specified by a tuple (m, ℓ, ε) where ℓ is a prime and $\varepsilon \in \{-1, 0, 1\}$, where we unify the notation by writing $(m, \ell, 0)$ in place of Weaver's (m, ℓ). Such a tuple corresponds to a family of congruences of the form (5.1) with coefficients

$$A = m\ell^{4-|\varepsilon|} \tag{5.2}$$

$$B = \frac{m\ell^{3-|\varepsilon|}\alpha + 1}{24} + m\ell^{3-|\varepsilon|}\delta, \tag{5.3}$$

where α is the unique solution of $m\ell^{3-|\varepsilon|}\alpha \equiv -1 \bmod 24$ with $1 \leqslant \alpha < 24$, and where $0 \leqslant \delta < \ell$ is any solution of

$$\begin{cases} 24\delta \not\equiv -\alpha \mod \ell & \text{if } \varepsilon = 0 \\ (24\delta + \alpha \mid \ell) = \varepsilon & \text{if } \varepsilon = \pm 1. \end{cases} \tag{5.4}$$

The free choice of δ gives $\ell - 1$ distinct congruences for a given tuple (m, ℓ, ε) if $\varepsilon = 0$, and $(\ell - 1)/2$ congruences if $\varepsilon = \pm 1$.

Weaver's test for congruence, described by [38, Theorems 7 and 8], essentially amounts to a single evaluation of $p(n)$ at a special point n. Namely, for given m, ℓ, we compute the smallest solutions of $\delta_m \equiv 24^{-1} \bmod m$, $r_m \equiv -m \bmod 24$, and check whether $p(mr_m(\ell^2 - 1)/24 + \delta_m)$ is congruent mod m to one of three values corresponding to the parameter $\varepsilon \in \{-1, 0, 1\}$. We give a compact statement of this procedure as Algorithm 5. To find new congruences, we simply perform a brute force search over a set of candidate primes ℓ, calling Algorithm 5 repeatedly.

Algorithm 5 Weaver's congruence test

Input: A pair of prime numbers $13 \leqslant m \leqslant 31$ and $\ell \geqslant 5$, $m \neq \ell$

Output: (m, ℓ, ε) defining a congruence, and Not-a-congruence otherwise

$\delta_m \leftarrow 24^{-1} \bmod m$ {Reduced to $0 \leqslant \delta_m < m$}

$r_m \leftarrow (-m) \bmod 24$ {Reduced to $0 \leqslant m < 24$}

$v \leftarrow \dfrac{m-3}{2}$

$x \leftarrow p(\delta_m)$ {We have $x \not\equiv 0 \bmod m$}

$y \leftarrow p\left(m\left(\dfrac{r_m(\ell^2-1)}{24}\right) + \delta_m\right)$

$f \leftarrow (3 \mid \ell)\,((-1)^v r_m \mid \ell)$ {Jacobi symbols}

$t \leftarrow y + fx\ell^{v-1}$

if $t \equiv \omega \bmod m$ where $\omega \in \{-1, 0, 1\}$ **then**

  return $(m, \ell, \omega\,(3(-1)^v \mid \ell))$

else

  return Not-a-congruence

end if

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

5.1. *Comparison of algorithms for vector computation*

In addition to the HRR formula, the author has added code to FLINT for computing the vector of values $p(0), p(1), \ldots, p(n)$ over $\mathbb{Z}$ and $\mathbb{Z}/m\mathbb{Z}$. The code is straightforward, simply calling the default FLINT routines for power series inversion over the respective coefficient rings, which in both cases invokes Newton iteration and FFT multiplication via Kronecker segmentation.

A timing comparison between the various methods for vector computation is shown in Table 3. The power series method is clearly the best choice for computing all values up to n modulo a fixed prime, having a complexity of $O(n^{1+o(1)})$. For computing the full integer values, the power series and HRR methods both have complexity $O(n^{3/2+o(1)})$, with the power series method expectedly winning.

Ignoring logarithmic factors, we can expect the HRR formula to be better than the power series for multi-evaluation of $p(n)$ up to some bound n when n/c values are needed. The factor $c \approx 10$ in the FLINT implementation is a remarkable improvement over $c \approx 1000$ attainable with previous implementations of the partition function. For evaluation mod m, the HRR formula is competitive when $O(n^{1/2})$ values are needed; in this case, the constant is highly sensitive to m.

For the sparse subset of $O(n^{1/2})$ terms searched with Weaver's algorithm, the HRR formula has the same complexity as the modular power series method, but as seen in Table 3 runs more than an order of magnitude faster. On top of this, it has the advantage of parallelizing trivially, being resumable from any point, and requiring very little memory (the power series evaluation mod $m = 13$ up to $n = 10^9$ required over 40 GB memory, compared to a few megabytes with the HRR formula). Euler's method is, of course, also resumable from an arbitrary point, but this requires computing and storing all previous values.

We mention that the authors of [7] use a parallel version of the recursive Euler method. This is not as efficient as power series inversion, but allows the computation to be split across multiple processors more easily.

5.2. *Results*

Weaver gives 167 tuples, or 76 065 congruences, containing all ℓ up to approximately 1000–3000 (depending on m). This table was generated by computing all values of $p(n)$ with $n < 7.5 \times 10^6$ using the recursive version of Euler's pentagonal theorem. Computing Weaver's table from scratch with FLINT, evaluating only the necessary n, takes just a few seconds. We are also able to numerically verify instances of all entries in Weaver's table for small k.

As a more substantial exercise, we extend Weaver's table by determining all ℓ up to 10^6 for each prime m. Statistics are listed in Table 4. The computation was performed by assigning subsets of the search space to separate processes, running on between 40 and 48 active cores for a period of four days, evaluating $p(n)$ at $6(\pi(10^6) - 3) = 470\,970$ distinct n ranging up to 2×10^{13}.

TABLE 3. *Comparison of time needed to compute multiple values of $p(n)$ up to the given bound, using power series inversion and the HRR formula. The rightmost column gives the time when only computing the subset of terms that are searched with Weaver's algorithm in the $m = 13$ case.*

n	Series ($\mathbb{Z}/13\mathbb{Z}$)	Series ($\mathbb{Z}$)	HRR (all)	HRR (sparse)
10^4	0.01 s	0.1 s	1.4 s	0.001 s
10^5	0.13 s	4.1 s	41 s	0.008 s
10^6	1.4 s	183 s	1430 s	0.08 s
10^7	14 s			0.7 s
10^8	173 s			8 s
10^9	2507 s			85 s

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

We find a total of 70 359 tuples, corresponding to slightly more than 2.2×10^{10} new congruences. To pick an arbitrary, concrete example, one 'small' new congruence is $(13, 3797, -1)$ with $\delta = 2588$, giving

$$p(711647853449k + 485138482133) \equiv 0 \mod 13$$

which we easily evaluate for all $k \leqslant 100$, providing a sanity check on the identity as well as the partition function implementation. As a larger example, $(29, 999\,959, 0)$ with $\delta = 999\,958$ gives

$$p(28995244292486005245947069k + 28995221336976431135321047) \equiv 0 \mod 29$$

which, despite our efforts, presently is out of reach for direct evaluation.

Complete tables of (ℓ, ε) for each m are available at:
http://www.risc.jku.at/people/fjohanss/partitions/
http://sage.math.washington.edu/home/fredrik/partitions/.

6. *Discussion*

Two obvious improvements to our implementation would be to develop a rigorous, and perhaps faster, version of Algorithm 3 for computing $\cos(p\pi/q)$ to high precision, and to develop fast multithreaded implementations of transcendental functions to allow computing $p(n)$ for much larger n. Curiously, a particularly simple AGM-type iteration is known for $\exp(\pi)$ (see [3]), and it is tempting to speculate whether a similar algorithm can be constructed for $\exp(\pi\sqrt{24n-1})$, allowing faster evaluation of the first term.

Some performance could also be gained with faster low-precision transcendental functions (up to a few thousand bits) and by using a better bound than (1.8) for the truncation error.

The algorithms described in this paper can be adapted to evaluation of other HRR-type series, such as the number of partitions into distinct parts

$$Q(n) = \frac{\pi^2\sqrt{2}}{24} \sum_{k=1}^{\infty} \frac{A_{2k-1}(n)}{(1-2k)^2}\, {}_0F_1\left(2, \frac{(n+\frac{1}{24})\pi^2}{12(1-2k)^2}\right). \tag{6.1}$$

Using asymptotically fast methods for numerical evaluation of hypergeometric functions, it should be possible to retain quasi-optimality.

Finally, it remains an open problem whether there is a fast way to compute the isolated value $p(n)$ using purely algebraic methods. We mention the interesting recent work by Bruinier and Ono [6], which perhaps could lead to such an algorithm.

Acknowledgements. The author thanks Silviu Radu for suggesting the application of extending Weaver's table of congruences, and for explaining Weaver's algorithm in detail. The author also thanks the anonymous referee for various suggestions, and Jonathan Bober

TABLE 4. *The number of tuples of the given type with $\ell < 10^6$, the total number of congruences defined by these tuples, the total CPU time, and the approximate bound up to which $p(n)$ was evaluated.*

m	$(m, \ell, 0)$	$(m, \ell, +1)$	$(m, \ell, -1)$	Congruences	CPU (h)	Max n
13	6 189	6 000	6 132	5 857 728 831	448	5.9×10^{12}
17	4 611	4 611	4 615	4 443 031 844	391	4.9×10^{12}
19	4 114	4 153	4 152	3 966 125 921	370	3.9×10^{12}
23	3 354	3 342	3 461	3 241 703 585	125	9.5×10^{11}
29	2 680	2 777	2 734	2 629 279 740	1 155	2.2×10^{13}
31	2 428	2 484	2 522	2 336 738 093	972	2.1×10^{13}
All	23 376	23 367	23 616	22 474 608 014	3 461	

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press

for pointing out that Erdős' theorem about quadratic nonresidues gives a rigorous complexity bound without assuming the Extended Riemann Hypothesis.

Finally, William Hart gave valuable feedback on various issues, and generously provided access to the computer hardware used for the large-scale computations reported in this paper. The hardware was funded by Hart's EPSRC Grant EP/G004870/1 (Algorithms in Algebraic Number Theory) and hosted at the University of Warwick.

References

1. N. Ankeny, 'The least quadratic non residue', *Ann. of Math.* (2) 55 (1952) no. 1, 65–72.
2. T. Apostol, *Modular functions and Dirichlet series in number theory*, 2nd edn (Springer, New York, 1997).
3. J. Borwein and D. Bailey, *Mathematics by experiment: plausible reasoning in the 21st century* (A K Peters, Wellesley, MA, 2003) 137.
4. J. Borwein and P. Borwein, *Experimental and computational mathematics: selected writings* (Perfectly Scientific Press, Portland, OR, 2010) 250.
5. R. Brent and P. Zimmermann, *Modern computer arithmetic* (Cambridge University Press, New York, 2011).
6. J. Bruinier and K. Ono, 'Algebraic formulas for the coefficients of half-integral weight harmonic weak Maass forms', Preprint, 2011, arXiv.org/abs/1104.1182.
7. N. Calkin, J. Davis, K. James, E. Perez and C. Swannack, 'Computing the integer partition function', *Math. Comp.* 76 (2007) no. 259, 1619–1638.
8. M. Cipolla, 'Un metodo per la risoluzione della congruenza di secondo grado', *Napoli Rend.* 9 (1903) 153–163.
9. R. Crandall and C. Pomerance, *Prime numbers: a computational perspective* (Springer, New York, 2005) 99–103.
10. P. Erdős, 'Remarks on number theory. I', *Mat. Lapok* 12 (1961) 10–17.
11. A. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier and P. Zimmermann, 'MPFR: A multiple-precision binary floating-point library with correct rounding', ACM Trans. Math. Software 33, no. 2, 2007, http://www.mpfr.org/.
12. The GMP development team, 'Computing billions of π digits using GMP', http://gmplib.org/pi-with-gmp.html.
13. The GMP development team, 'GMP: the GNU multiple precision arithmetic library', http://gmplib.org/.
14. P. Hagis Jr, 'A root of unity occurring in partition theory', *Proc. Amer. Math. Soc* 26 (1970) no. 4, 579–582.
15. G. H. Hardy and S. Ramanujan, 'Asymptotic formulae in combinatory analysis', *Proc. Lond. Math. Soc.* 17 (1918) 75–115.
16. W. Hart, 'Fast library for number theory: an introduction', *Mathematical software – ICMS 2010*, Lecture Notes in Computer Science 6327, 88–91, http://www.flintlib.org.
17. W. Hart, 'A one line factoring algorithm', *J. Aust. Math. Soc.* 92 (2012) 61–69.
18. N. Higham, *Accuracy and stability of numerical algorithms*, 2nd edn (SIAM, Philadelphia, 2002).
19. Intel corporation, *Pentium processor family developer's manual. Volume 3: architecture and programming manual*, 1995, http://www.intel.com/design/pentium/MANUALS/24143004.pdf.
20. D. Knuth, 'Notes on generalized Dedekind sums', *Acta Arith.* 33 (1977) 297–325.
21. D. Knuth, *Fascicle 3: generating all combinations and partitions*, The Art of Computer Programming, vol. 4 (Addison-Wesley, 2005).
22. D. Lehmer, 'On the series for the partition function', *Trans. Amer. Math. Soc.* 43 (1938) no. 2, 271–295.
23. D. Lehmer, 'On a conjecture of Ramanujan', *J. Lond. Math. Soc.* 11 (1936) 114–118.
24. D. Lehmer, 'On the Hardy–Ramanujan series for the partition function', *J. Lond. Math. Soc.* 3 (1937) 171–176.
25. The MPIR development team, 'MPIR: multiple precision integers and rationals', http://www.mpir.org.
26. A. Odlyzko, 'Asymptotic enumeration methods', *Handbook of combinatorics 2* (eds R. Graham, M. Grötschel and L. Lovász; Elsevier, The Netherlands, 1995) 1063–1229, http://www.dtc.umn.edu/∼odlyzko/doc/asymptotic.enum.pdf.
27. OEIS Foundation Inc, 'The on-line encyclopedia of integer sequences', 2011, http://oeis.org/A000041.
28. K. Ono, 'The distribution of the partition function modulo m', *Ann. of Math.* (2) 151 (2000) 293–307.
29. The Pari/GP development team, 'Pari/GP, Bordeaux', 2011, http://pari.math.u-bordeaux.fr/.
30. P. Pollack, 'The average least quadratic nonresidue modulo m and other variations on a theme of Erdős', *J. Number Theory* 132 (2012) no. 6, 1185–1202.
31. H. Rademacher, 'On the partition function $p(n)$', *Proc. Lond. Math. Soc.* 43 (1938) 241–254.
32. H. Rademacher and A. Whiteman, 'Theorems on Dedekind sums', *Amer. J. Math.* 63 (1941) no. 2, 377–407.
33. D. Shanks, 'Five number-theoretic algorithms', *Proceedings of the Second Manitoba Conference on Numerical Mathematics*, 1972, 51–70.

34. W. Stein and The Sage development team, 'Sage: open source mathematics software', http://www.sagemath.org.
35. Sun Microsystems Inc, 'FDLIBM version 5.3', http://www.netlib.org/fdlibm/readme.
36. A. Tonelli, 'Bemerkung über die Auflösung quadratischer Congruenzen', *Göttinger Nachrichten* (1891) 344–346.
37. W. Watkins and J. Zeitlin, 'The minimal polynomial of $\cos(2\pi/n)$', *Amer. Math. Monthly* 100 (1993) no. 5, 471–474.
38. R. Weaver, 'New congruences for the partition function', *J. Ramanujan* 5 (2001) 53–63.
39. A. Whiteman, 'A sum connected with the series for the partition function', *Pacific J. Math.* 6 (1956) no. 1, 159–176.
40. Wolfram Research Inc., 'Some notes on internal implementation', Mathematica documentation center, 2011, http://reference.wolfram.com/mathematica/note/SomeNotesOnInternalImplementation.html.
41. P. Zimmermann, *The bit-burst algorithm*, Slides presented at computing by the numbers: algorithms, precision, and complexity, Berlin, 2006, http://www.loria.fr/∼zimmerma/talks/arctan.pdf.

Fredrik Johansson
Research Institute for Symbolic Computation
Johannes Kepler University
Altenberger Strasse 69, 4040 Linz
Austria

fredrik.johansson@risc.jku.at

https://doi.org/10.1112/S1461157012001088 Published online by Cambridge University Press