Fast Primality Testing for Integers That Fit into a Machine Word*

Michal Forišek and Jakub Jančina

Comenius University, Bratislava, Slovakia,
forisek@dcs.fmph.uniba.sk, jakub.jancina@gmail.com

Abstract. For large integers, the most efficient primality tests are probabilistic. However, for integers with a small fixed number of bits the best tests in practice are deterministic. Currently the best known tests of this type involve 3 rounds of the Miller-Rabin test for 32-bit integers and 7 rounds for 64-bit integers. Our main result in this paper: For 32-bit integers we reduce this to a single computation of a simple hash function and a single round of Miller-Rabin. Similarly, for 64-bit integers we can reduce the number of rounds to two (but with a significantly large precomputed table) or three. Up to our knowledge, our implementations are the fastest one-shot deterministic primality tests for 32-bit and 64-bit integers to date.

We also provide empirical evidence that our algorithms are fast in practice and that the data segment is roughly as small as possible for an algorithm of this type.

1 Overview

In this section we give an overview of relevant topics related to primality testing.

1.1 Large Primes, Deterministic Tests

Since the seminal result by Agrawal, Kayal, and Saxena in 2002, we know that primality testing is in P. Their original paper [1] gives an algorithm that runs in $\tilde{O}(\log^{12} n)$ when testing whether n is a prime.[1] A more recent paper by Pomerance and Lenstra [10] improves this to $\tilde{O}(\log^{6} n)$. While certainly interesting from a theoretical point of view, these algorithms are rarely used in practice. For instance, in most cryptographic applications primes with $\log_2 n \approx 1000$ are quite common, and the known deterministic tests are way too slow for numbers of this magnitude.

* The research is partially funded by the VEGA grant V-12-031-00.

[1] Here, $\tilde{O}(f(n))$ formally denotes the class $\bigcup_k O(f(n) \log^k f(n))$.

1.2 Large Primes, Probabilistic Tests

For large primes, the state of the art are general unconditional probabilistic tests. Here, “general” means that they work for all possible n, and “unconditional” means that their correctness does not depend on any unproved hypothesis (such as the generalized Riemann hypothesis). Most of these tests only have a one-sided error: for prime n they always report the correct answer, but for composite n they have a small probability of incorrectly reporting that n is prime.

The two most notable tests with this property are the Solovay-Strassen test ([14], Theorem 1 below) and the Miller-Rabin test ([11, 13], Theorem 2 below).

Theorem 1 (Solovay, Strassen). *For a given odd integer $n > 3$:*
Choose an integer a uniformly at random from[2] $[2, n-1]$.
Compute $x = a^{(n-1)/2} \bmod n$, and the Jacobi symbol for a and n: $j = \left(\frac{a}{n}\right)$.
If n is prime, we always have $j \neq 0$ and $x = j$.
If n is composite, with probability at least 1/2 we have $j = 0$ or $x \neq j$.

Definition 1. *Let $n - 1 = 2^s d$ with d odd and s non-negative. The positive integer n is called a* strong probable-prime *with base b (b-SPRP) if either $b^d \equiv 1 \pmod{n}$, or $(b^d)^{2^r} \equiv -1 \pmod{n}$ for some $r \in [0, s-1]$.*

If n is a composite b-SPRP, we call it a strong pseudoprime *with base b.*

Theorem 2 (Rabin, Monier). *For a given odd integer $n > 3$:*
Choose an integer b uniformly at random from $[2, n-2]$.
If n is prime, then n is also a b-SPRP.
If n is composite, with probability at least 3/4 it is not a b-SPRP.

The Solovay-Strassen test can be implemented in $O(\log^3 n)$ by using exponentiation by squaring and the law of quadratic reciprocity to compute the Jacobi symbol. The Miller-Rabin test can be implemented in $\tilde{O}(\log^2 n)$ using FFT-based multiplication of big integers.

Note that each test can be repeated arbitrarily many times: e.g., k independent runs of the Miller-Rabin test decrease the probability of error to 4^{-k}.

1.3 Small Primes, Applications

Certainly the most famous application of primality testing lies in cryptography, where we need to test integers that are as large as possible. However, primality testing also has many other uses. And for many of those use cases the integers we need to test for primality come from a bounded range. Here we shall mention a few such applications:

- Data structures based on hashing (esp. universal hashing).
- Prime moduli for pseudo-random number generators.
- Random primes used as a tool in various other probabilistic algorithms, e.g. polynomial identity testing.

[2] We use the interval notation $[a, b]$ to denote the set of integers $\{a, a+1, \dots, b\}$.

In all these cases, all integers we work with fit into a standard machine word, which is nowadays usually either a 32-bit or a 64-bit integer variable. As shown below, the general probabilistic tests are not the most efficient ones in this setting.

1.4 Small Primes, Deterministic Tests

The basic deterministic primality test is trial division: given an n, we compute $n \bmod d$ for all $d \in [2, \lfloor\sqrt{n}\rfloor\,]$. The disadvantages are obvious: its time complexity is not polynomial in the input size, which makes the algorithm usable only for very small values of n.

Some tools used in practice opt to use probabilistic tests also for small integers. For instance, the current implementation of GNU `factor` uses a Las Vegas (i.e., zero-error) probabilistic algorithm that alternates between running a round of the Miller-Rabin test to verify compositeness and a round of the Lucas primality test (see [4]) to verify primality. Note that the Lucas test is impractical in general for large primes, as it requires the factorization of $n-1$.

The state of the art are exact deterministic tests based on SPRP testing: if we have a limited range of valid inputs, we can easily turn the probabilistic Miller-Rabin test into a deterministic one by choosing a fixed set of bases that, taken together, detects all the composite numbers within our range.

Currently, the best known test with two bases due to Izykowski and Panasiuk [7] works until $n = 1\,050\,535\,501$. Hence, the best known tests for $n < 2^{32}$ have three bases. The first such set was found by Jaeschke [8]. The best known set for $n < 2^{64}$ was found by Sinclair in 2011 (unpublished, verified by Izykowski) and has seven bases. These two sets are presented below in Theorem 3. See also [12, 15] for older related results and [6] for an up-to-date collection of other known records.

Theorem 3 (Jaeschke, Sinclair). *If $n < 2^{32}$ is a b-SPRP for $b \in \{2, 7, 61\}$, then n is a prime. If $n < 2^{64}$ is a b-SPRP for $b \in \{2, 325, 9375, 28178, 450775, 9780504, 1795265022\}$, then n is a prime.*

When testing primality of an n that fits into a machine word (e.g., a 32-bit or a 64-bit unsigned integer in our case), it is perhaps best to estimate the running time of a single SPRP test as $O(\log n)$, as said test involves $O(\log n)$ elementary multiplications and modulo operations.

1.5 Baillie-PSW Test

To conclude our overview, we dedicate a separate section to the Baillie-PSW test [2]: a combination of the 2-SPRP test and another similar test (the Lucas probable prime test[3]). This is a deterministic test. It is known that the test is correct for all $n < 2^{64}$. Currently, there are no known counterexamples for larger n, but there is a heuristic argument that they should exist.

[3] This is a different test from the Lucas primality test mentioned above. This test may err in the opposite direction: by claiming that a composite number is prime.

2 Towards a Faster Primality Test for 32-bit/64-bit Integers

Instead of having to test three different SPRP bases in the worst case, we designed a faster class of algorithms that work as follows:

1. Use trial division to check that n is relatively prime to 210.
2. In constant time, compute a hash value $h(n)$.
3. Use a pre-computed lookup table to determine a single base $b_{h(n)}$ such that n is a $b_{h(n)}$-SPRP iff n is a prime.

Note that the first step (trial division by 2, 3, 5, and 7) is also included in other practical primality testing algorithms, as it always decreases the average case by a constant factor. As we show below, the constant time spent on the hashing and lookup is negligible in comparison to a single SPRP test, which makes our algorithm faster in practice.

As $\phi(210) = 48$, the trial division reduces the search space to $48/210 \approx 22.86\%$. We will use M_{32} and M_{64} to denote the set of 32-bit/64-bit numbers that are relatively prime to 210.

In the following two sections, we present our approximate calculations that were used to select the right parameters for our search. The actual algorithms are then presented in later sections of this paper.

2.1 Heuristic Arguments about Hash Table Size, 32-bit Case

When trying to find a suitable hash function and a set of bases, as described above, we should aim to minimize the size of the precomputed lookup table. This is because that table has to be loaded into memory when our program starts. Also, keeping the table small makes the algorithm more cache-friendly and makes the calls to our primality test function faster in practice.

When minimizing the hash table size, we are facing two limits. First of all, the size of the hash table has to be large enough for the bases to *exist*. But that's not enough. Verifying a base requires a considerable amount of computation,[4] and we must be able to *find* good bases using our limited resources.

Already a single SPRP test gives us a lot of information – for example, in M_{32} there are only about 2000 strong pseudoprimes for any prime base. Exact counts for some prime bases are given in Appendix A, the mean of known values is $\mu \approx 2152$. That is, for any fixed prime base b, if we choose an $n \in M_{32}$ uniformly at random, the probability that it is a base-b strong pseudoprime is only about $p \approx 2152/|M| \approx 0.0000022$.

[4] When searching for the best hash function, we used two different implementations. A standard desktop computer is capable of about 1M SPRP tests per second. Later we switched to a CUDA-based implementation that was approximately 10 times more efficient.

By choosing a good hash function, we are partitioning M_{32} into buckets of approximately the same size. Suppose that we have a bucket with s elements. The probability that a fixed base b correctly classifies all s elements in the bucket can now be estimated well by the value $q_1 = (1-p)^s$.

For our estimate of p, we get $q_1 \approx 2.3 \cdot 10^{-15}$ for a hash function with 64 buckets, $q_1 \approx 4.8 \cdot 10^{-8}$ for a hash function with 128 buckets, and $q_1 \approx 0.000219$ for 256 buckets. Thus, it's basically impossible to have 64 or fewer buckets, and it will be quite hard to get 128 or fewer.

We would now like to extend this estimate to multiple bases. We could now be tempted to make the following *incorrect* conclusion: "For a fixed bucket, each particular base fails with probability $1 - q_1$, hence if we test k different bases, we expect all of them to fail with probability $(1-q_1)^k$". The above claim is incorrect because the distributions of strong pseudoprimes for different bases are not independent.

More precisely, there is a very strong positive correlation – if a number is a strong pseudoprime in one base, the probability that it is a pseudoprime in a different base is significantly greater than an independent distribution would suggest. If the distributions were independent, for any two bases b_1 and b_2 we would have expected to have $p^2|M_{32}| \approx 0.00474$ numbers that are strong pseudoprimes in both bases. However, in practice there are, on average, about 100 such numbers. (Again, please refer to Appendix A for exact measured data.) In other words, if one particular base does not work, chances are that other bases won't work as well.

How can we deal with this issue? It turns out that once we pick *six* different bases, they are very likely to have no strong pseudoprimes in common in M_{32}. (Here, 6 is the smallest positive integer with this property.) Hence, for a decent estimate of what happens when testing multiple bases we can divide the bases into disjoint groups of five and treat them as independent. Thus, all we need now is an estimate of the probability q_5 that at least one of five prime bases works for a given bucket. We can then reasonably approximate the probability of any of $5k$ bases working as $1 - (1-q_5)^k$.

Additionally, if k is significantly larger than the number of buckets we have, we may treat the buckets as independent. Hence, the probability that for each of h buckets we can find a good base is approximated well by $(1-(1-q_5)^k)^h$.

Given the approximate counts of strong pseudoprimes with two to five different bases at the same time, we can easily compute q_5 using the principle of inclusion and exclusion. For our p, we get $q_5 \approx 8.8 \cdot 10^{-15}$ for 64 buckets, $q_5 \approx 2.25 \cdot 10^{-7}$ for 128 buckets, and $q_5 \approx 0.001077$ for 256 buckets. (Note that these probabilities are somewhat smaller than the ones we would get for an independent distribution.)

Given the size of M_{32}, we can realistically hope to be able to process approximately 10 000 different bases. Thus, we decided to cap our search to 16-bit bases. For these, we get that the probabilities of successfully finding the entire table of bases is essentially zero for 128 buckets, and essentially one for 256 buckets.

The threshold appears to be somewhere around 220 buckets: our estimate gives the probability about 0.15% for 220 buckets but about 36.3% for 230 buckets. Note that the smallest hash function we actually found has 224 buckets, which matches these estimates nicely.

2.2 Heuristic Arguments About Hash Table Size, 64-bit Case

The most useful resource when dealing with the 64-bit case is the table of all base-2 strong pseudoprimes up to 2^{64}. We acquired this list from a larger data set computed by Feistma [5].

There are 31 894 014 base-2 strong pseudoprimes in our range (and we have good statistical reasons to believe that the number of pseudoprimes for other bases is similar). We will denote this set S_2.

First of all, we should immediately realize that this huge number basically rules out any reasonable chance of an algorithm of our type with just one SPRP test. With about $32M$ bad numbers, we would need a huge number of buckets in order to have a decent chance that a fixed bucket received no bad numbers.

Also, as we could not afford repeating and extending the computation done by Feistma for the entire range up to 2^{64}, we opted to use a different approach: after the trial division, our algorithm for 64-bit numbers will always perform a base-2 SPRP test. This will leave us with just the $32M$ known numbers as false positives, and we will handle those with a hash function and another SPRP test (or two).

At a first glance, finding a suitable hash function seems easier than in the 32-bit case – after all, $|S_2|$ is about $1/24$ of $|M_{32}|$. However, these are all numbers known to be base-2 strong pseudoprimes, and therefore they are much more likely to be strong pseudoprimes in other bases as well. For example, 1 501 720 out of the numbers in S_2 are also base-3 strong pseudoprimes. This makes finding a small hash function significantly harder.

Computations similar to the 32-bit case can be used to show that already with 400 numbers per bucket it is 99.9% certain that no base up to 2^{17} will work for a given bucket. Thus, the best we can hope for if we want a two-test algorithm is one where the hash table has more than $|S_2|/400 \approx 100,000$ buckets. (It might be possible to get a smaller table by considering more bases, but our estimates show that even if we could process all bases up to 2^{64}, we would still require tens of thousands of buckets.)

The best two-test algorithm found by our search has $2^{18} = 262,144$ buckets, and each of the precomputed bases has at most 17 bits. Our estimates above show that while this can be improved, the improvement will not be significant. Still, even with a table this large, the algorithm turns out to be fast in practical tests.

One possibility how to decrease the size of the precomputed table is a trade-off: we can decrease it by increasing the number of Miller-Rabin tests from two to three. In other words, we will always have the initial base-2 test, followed by two other tests that are different for each bucket.

3 An Overview of the New Algorithms

In this section we present actual algorithms we constructed for the 32-bit and 64-bit cases, according to the analysis presented above. We will start by listing a full implementation of a correct and fast primality test for 32-bit integers. The precomputed hash table contains 256 small integers and fits conveniently into 512 bytes of memory. This is the version we recommend to be used for 32-bit integers. Below, we refer to this algorithm as `FJ32_256`.

```
uint16_t bases[]={15591,2018,166,7429,8064,16045,10503,4399,1949,1295,2776,3620,560,3128,5212,
2657,2300,2021,4652,1471,9336,4018,2398,20462,10277,8028,2213,6219,620,3763,4852,5012,3185,
1333,6227,5298,1074,2391,5113,7061,803,1269,3875,422,751,580,4729,10239,746,2951,556,2206,
3778,481,1522,3476,481,2487,3266,5633,488,3373,6441,3344,17,15105,1490,4154,2036,1882,1813,
467,3307,14042,6371,658,1005,903,737,1887,7447,1888,2848,1784,7559,3400,951,13969,4304,177,41,
19875,3110,13221,8726,571,7043,6943,1199,352,6435,165,1169,3315,978,233,3003,2562,2994,10587,
10030,2377,1902,5354,4447,1555,263,27027,2283,305,669,1912,601,6186,429,1930,14873,1784,1661,
524,3577,236,2360,6146,2850,55637,1753,4178,8466,222,2579,2743,2031,2226,2276,374,2132,813,
23788,1610,4422,5159,1725,3597,3366,14336,579,165,1375,10018,12616,9816,1371,536,1867,10864,
857,2206,5788,434,8085,17618,727,3639,1595,4944,2129,2029,8195,8344,6232,9183,8126,1870,3296,
7455,8947,25017,541,19115,368,566,5674,411,522,1027,8215,2050,6544,10049,614,774,2333,3007,
35201,4706,1152,1785,1028,1540,3743,493,4474,2521,26845,8354,864,18915,5465,2447,42,4511,
1660,166,1249,6259,2553,304,272,7286,73,6554,899,2816,5197,13330,7054,2818,3199,811,922,350,
7514,4452,3449,2663,4708,418,1621,1171,3471,88,11345,412,1559,194};

bool is_SPRP(uint32_t n, uint32_t a) {
    uint32_t d = n-1, s = 0;
    while ((d&1)==0) ++s, d>>=1;
    uint64_t cur = 1, pw = d;
    while (pw) {
        if (pw & 1) cur = (cur*a) % n;
        a = ((uint64_t)a*a) % n;
        pw >>= 1;
    }
    if (cur == 1) return true;
    for (uint32_t r=0; r<s; r++) {
        if (cur == n-1) return true;
        cur = (cur*cur) % n;
    }
    return false;
}

bool is_prime(uint32_t x) {
    if (x==2 || x==3 || x==5 || x==7) return true;
    if (x%2==0 || x%3==0 || x%5==0 || x%7==0) return false;
    if (x<121) return (x>1);
    uint64_t h = x;
    h = ((h >> 16) ^ h) * 0x45d9f3b;
    h = ((h >> 16) ^ h) * 0x45d9f3b;
    h = ((h >> 16) ^ h) & 255;
    return is_SPRP(x,bases[h]);
}
```

Two other algorithms that use different constants and a different hash function are published in [9]. One of those has only 224 buckets for its hash function, the other has 1024 but each base has only up to 8 bits, and the test uses a yet-faster hash function. Below, we refer to these as `FJ32_224` and `FJ32_1024`.

For obvious reasons, we decided not to include listings of the algorithms for the 64-bit case. We have two sample implementations. One of them (`FJ64_262k`) is the 2-test algorithm mentioned above, with $262,144$ buckets in its hash function and at most 16-bit bases. The other (`FJ64_16k`) is a 3-test version with only $16,384$ buckets and at most 12-bit bases.

Digital versions of all these algorithms (incl. precomputed tables) are available online at `http://people.ksp.sk/~misof/primes/`

3.1 A Note on the Choice of the Hash Function

The choice of the particular hash function used in the algorithm does not actually matter much. A similar construction should be possible with any hash function that distributes its inputs in an approximately uniform way.

When searching for the actual algorithms listed above, we simply picked a class of hash functions that can be computed in constant time and tend to distribute the inputs in a sufficiently uniform way. Of course, the choice of a particular hash function is easily validated post hoc by actually finding a working set of bases of a sufficiently small size.

4 Theoretical Analysis if Our Algorithms

In this section we provide a theoretical comparison of our algorithms with the 3-base algorithm by Jaeschke (in the 32-bit case) and the 7-base algorithm by Sinclair (in the 64-bit case).

We will consider both their worst-case and average-case performance. Here, the average is taken over all valid inputs – in other words, we are talking about their expected running time for a randomly chosen input.

Obviously, the running time of all considered algorithms is dominated by the number of SPRP tests performed. Already for $n \in [0, 2^{32}-1]$, the average number of multiply-and-modulo operations in a single SPRP test is approximately 60. Both the trial divisions and the computation of our hash function are negligible in comparison.

Hence, the worst-case running time of our algorithms `FJ32_*` should be approximately 3 times better than Jaeschke in the 32-bit case. In the 64-bit case, `FJ64_262k` should be about 3.5 times, and `FJ64_16k` about 2.3 times faster than Sinclair.

Now let's consider the average case. The old algorithms may sometimes perform more than one SPRP test. For example, the Jaeschke algorithm performs two or three tests when n is a base-2 strong pseudoprime. However, for a lower bound on their average case we may simply ignore the pseudoprimes and focus on the worst inputs: primes.

There are $\pi(n) \approx n/\ln n$ primes smaller than n. Thus, there are $|M_{32}|$ inputs where the Jaeschke algorithm performs at least one SPRP test, and out of those at least $\pi(2^{32}) = 203,280,221$ such that it performs at least three tests. Thus, the expected number of SPRP tests performed for a random $n \in [0, 2^{32}-1]$ by the Jaeschke algorithm is at least 0.323. On the other hand, our algorithms `FJ32_*` perform exactly one SPRP test for each input from M_{32}, which makes their expected number of SPRP tests only 0.229.

This makes our algorithms about 1.4 times faster in the average case.

We can make a similar argument in the 64-bit case. We have $\pi(2^{64}) \approx 4.158 \cdot 10^{17}$. Sinclair's algorithm does at least one SPRP test for each number in M_{64}, and all seven for each prime. Hence, the expected number of SPRP tests in the Sinclair algorithm is at least 0.3638. On the other hand, our algorithms do at most two/three tests for each prime and each base-2 strong pseudoprime, and at most one test for each other number in M_{64}. Hence, the expected number of SPRP tests is at most 0.274 for `FJ64_262k` and at most 0.2962 for `FJ64_16k`.

Hence, we should expect `FJ64_262k` to be about 1.33 times, and `FJ64_16k` to be about 1.23 times faster than Sinclair in the average case.

5 Practical Tests of Our Algorithms

We tested our algorithms against various other implementations. We used our implementation of the Jaeschke algorithm in the 32-bit, and of the Sinclair algorithm (both with trial division by 2, 3, 5, and 7) in the 64-bit case. Additionally, we used the following other implementations:

- `TwoBase`: a 2-SPRP and a 15-SPRP test, followed by a binary search check whether the input is one of 59 counterexamples (only used for 32 bits).
- `MPZ`: a 25-round probabilistic Miller-Rabin test implemented by the call of a GMP library function `mpz_probab_prime_p(x,25)`.
- `BPSW`: Baillie-PSW test implementation by Cleaver [3].
- `PrimeQ`: Primality test in Mathematica.

We used the Mersenne Twister implementation in the g++ `<random>` library to generate uniformly distributed numbers for testing. When testing the average case, we generated 10^7 numbers and tested each for primality once. When testing the worst case, we generated at least 10^4 numbers and tested each 10^3 times in order to get more precise measurements.

All test were performed on a 64-bit Athlon processor running Linux.

The results of these tests are summarized in the tables below. The value in the "average" column is the time in milliseconds needed to process the entire data set. The value in the "worst" column is the time in microseconds that is needed to test a single number. (This value is computed as the maximum taken over all tested numbers of the average time spent on a single execution of the primality test.)

algorithm	average (ms)	worst (μs)
FJ32_1024	1000	2.423
FJ32_256	1029	2.418
FJ32_224	1038	2.413
TwoBase	1234	3.090
Jaeschke	1460	4.328
MPZ	21427	59.822
BPSW	27195	26.959
PrimeQ	49281	—

algorithm	average (ms)	worst (μs)
`FJ64_262k`	7660	15.755
`FJ64_16k`	9688	22.337
`Sinclair`	10896	53.184
`BPSW`	44539	68.192
`MPZ`	30831	149.042
`PrimeQ`	79202	—

As predicted by the analysis in the previous section, our new algorithms outperform the rest in all tests. The differences in runtime between our tests and Jaeschke/Sinclair roughly correspond to the theoretical predictions as well.

5.1 A Note on Precomputing the Table of Primes

All the above algorithms are designed as one-shot algorithms that are fast to execute without the need for any precomputation *during the actual execution of the algorithm.* Even though the fastest ones presented above are the fastest ones currently known for this type of usage, there is a related setting where this is not true.

If we expect that we'll need to test a significantly large number of small integers for primality, the best solution might be precomputing all possible answers and then answering each query in constant time. The canonical implementation uses one bit for each odd number, i.e., $2^b/16$ bytes of memory if the valid inputs are b-bit integers. This evaluates to approx. 268 MB of memory for 32-bit integers. This is obviously impractical for a one-shot test (even loading the 268 MB of data into memory is way too slow) but once the data is loaded the simple lookup outperforms even the one-round Miller-Rabin test significantly.

6 Conclusions

We have presented what we believe to be the most efficient algorithms to date to check the primality of 32-bit and 64-bit integers.

In the 32-bit case, we recommend using the algorithm `FJ32_256`. Based on our analysis, we expect this algorithm to be pretty close to being optimal, both in terms of the size of the precomputed data and running time. In other words: it is possible that significantly more efficient tests exist, but they have to be based on a different approach.

In the 64-bit case, we recommend using the algorithm `FJ64_262k`.

We expect that `FJ64_16k` is still quite far from being optimal. That is, it should be possible to find a three-base test for 64-bit numbers with a significantly smaller hash table size. We leave that as an open question for future research.

A Strong Pseudoprimes up to 2^{32}

There are 2256 base-2 strong pseudoprimes in M_{32}. The following table lists the number of base-b strong pseudoprimes in M_{32} for all prime b smaller than 100.

```
 3:  2680    5:  2269    7:  2053   11:  1953   13:  1965   17:  2026   19:  2071   23:  1936
29:  2005   31:  1965   37:  1899   41:  1976   43:  1978   47:  1957   53:  1959   59:  2057
61:  1873   67:  1985   71:  1951   73:  1846   79:  2126   83:  1953   89:  2003   97:  2000
```

We also tested 200 random prime bases smaller than 2^{32}. The number of strong pseudoprimes had a sample mean of 2152.7, a sample stdev of 129.9. Thus, we may expect most values for other bases to lie in the 2σ interval of $[1892.9, 2412.5]$.

We also tested about 1000 pairs of bases, each time computing the count of numbers that are strong pseudoprimes with both bases. For these counts, the sample mean was 100.75, and the sample stdev was 14.27. (The minimum and maximum encountered were 63 and 162, respectively.)

References

1. Agrawal, M., Kayal, N. and Saxena, N.: PRIMES in P. Ann. of Math., 160, 781–793 (2004)
2. Baillie, R. and Wagstaff, Jr., S.S.: Lucas Pseudoprimes. Mathematics of Computation, 35, 1391–1417 (1980)
3. Cleaver, D.: Baillie-Pomerance-Selfridge-Wagstaff test implementation (mpz prp.c). http://sourceforge.net/projects/mpzprp/files/ (2013)
4. Crandall, R. and Pomerance, C.: Prime Numbers: a Computational Perspective. Springer, 2nd edition (2005)
5. Feistma, J.: List of pseudoprimes and their prime factorizations, with additional annotations. http://www.cecm.sfu.ca/Pseudoprimes/index-2-to-64.html (2013)
6. Izykowski, W.: The best known SPRP bases sets. https://miller-rabin.appspot.com/ (2014)
7. Izykowski, W. and Panasiuk, M.: Finding strong probable prime bases for efficient ranged primality. Technical report, http://priv.ckp.pl/wizykowski/sprp.pdf (2011)
8. Jaeschke, G.: On strong pseudoprimes to several bases. Mathematics of Computation, 61, 915–926 (1993)
9. Jančina, J.: Rýchle testy prvočíselnosti pre obmedzený rozsah vstupov (Fast primality tests for a limited range of inputs, in Slovak). Bachelor thesis at Comenius University (2014)
10. Lenstra, H.W. and Pomerance, C.: Primality testing with Gaussian periods. Technical report, http://www.math.dartmouth.edu/~carlp/aks041411.pdf (2011)
11. Monier, L.: Evaluation and comparison of two efficient probabilistic primality testing algorithms. Theoretical Computer Science, 12, 97–108 (1980)
12. Pomerance, C., Selfridge, J.L. and Wagstaff, Jr., S.S.: The pseudoprimes up to $25 \cdot 10^9$. Math. Comp., 35, 1003–1026 (1980)
13. Rabin, M.O.: Probabilistic algorithm for testing primality. Journal of number theory, 12, 128–138 (1980)
14. Solovay, R. and Strassen, V.: A fast Monte-Carlo test for primality. SIAM journal on Computing, 6, 84–85 (1977)
15. Worley, S.: Optimization of Primality Testing Methods by GPU Evolutionary Search. Technical report,http://www.gpgpgpu.com/gecco2009/6.pdf (2009)