Baillie - PSW Primality Test
Variant Algorithm

Jim Cullen, February 2021

The Baillie–PSW primality test is a probabilistic primality testing algorithm, named after Robert Baillie, Carl Pomerance, John Selfridge, and Samuel Wagstaff. I tend to think of such algorithms as *composite proving* algorithms. They apply one or more criteria to prove that a candidate number **N** is composite. In the case of the criteria failing to prove compositeness, the number in question is assumed to be a prime number, with some level of probability.

While not a deterministic algorithm, the Baillie-PSW test is an extremely strong probabilistic test, with no known pseudoprimes and is in fact deterministic for any candidate less than 2^{64} or about 1.8447×10^{19}. The Baillie-PSW test is a combination of a Miller-Rabin prime test base 2 followed by a strong Lucas prime test. Both tests have their own pseudoprimes with no known overlap between them as they tend to be different residues **Mod N**.

Basics of the Baillie-PSW Test

Given a candidate number **N**, a Miller-Rabin test is performed to base 2. If **N** is not shown to be composite, proceed with the Strong Lucas Test. I first check that **N** is not a perfect square, to avoid an issue with the selection of the determinant parameter **D** described below. **N** being a perfect square would also be proof of compositeness. Next, according to Selfridge's *Method A*, a determinant **D** is selected from

$$\mathbf{\{ 5, -7 , 9 , -11 , 13 , -15 , \ldots \}}$$

such that **Jacobi_Symbol (D,N)** is found to be equal to **-1**. If **N** is a perfect square then no suitable **D** will ever be found. If the result of **Jacobi_Symbol (D,N)** is found to equal zero, then **D** must divide **N**. In that case, **N** must have factors and the test ends with **N** having been proven composite.

With a suitable value for **D** having been found, other parameters are set in preparation for the calculation of two Lucas sequences, $\mathbf{U_n}$ and $\mathbf{V_n}$. Specifically,

$$\mathbf{P = 1} \text{ and } \mathbf{Q = (1-D)/4}$$
$$\mathbf{U_0 = 0} \text{ and } \mathbf{V_0 = 2}$$

A set of binary doubling formulas are then applied to concurrently calculate the terms of both the $\mathbf{U_n}$ and $\mathbf{V_n}$ sequences, **Mod N**, up to **N+1** terms. Finally we do a simple congruency check to determine the result of the test. If

$$\mathbf{U_{n+1} = 0 \text{ Mod } N}$$

then **N** is extremely likely to be prime. There are alternate methods in use for determining the values of the initial parameters, and there are additional congruency checks to strengthen the test, but the above description is the one most commonly used. It's worth noting that even the most basic implementation with just the one congruency check given above, has no known pseudoprimes.

Variation of the Baillie-PSW Test

The variation I have come up with solves two problems with the usual implementation of the Baillie-PSW Primality Test. First is the fact that two separate sequences must be calculated, $\mathbf{U_n}$ and $\mathbf{V_n}$. Secondly, the calculation of those sequences, using doubling formulas, requires the left to right bits of the binary representation of **N**. This is somewhat tedious and time consuming to implement.

I begin with a simple prime product **gcd** sieve in two steps to check for prime factors between **2** and **97**, not because it is necessary, but to increase the efficiency of the algorithm time-wise. Some may argue that a sieve for more than 25 prime factors would be more appropriate, but the algorithm is fast enough that checking for divisibility by the first 25 primes seemed a good compromise. This step was omitted during testing to better determine the strength of my Baillie-PSW testing variation.

The Miller-Rabin Test to base 2, a probabilistic primality test, is then performed in the classic manner. This test is very powerful and eliminates many candidates, saving time by requiring only a few candidates to be subjected to the Strong Lucas Test, the most time intensive portion of the testing.

Eliminating perfect squares is then accomplished immediately using a brute force method. If $\mathbf{floor(\sqrt{N})^2 = N}$, then **N** is declared composite and the test ends. This step also facilitates the selection of determinant **D** in the Strong Lucas Test, which prohibits **N** being a perfect square.

Next is the Strong Lucas Test, which begins with the usual parameter selection, plus another parameter I've added, **R**. According to Selfridge's *Method A*, a determinant **D** is selected from

$$\mathbf{\{ 5, -7 , 9 , -11 , 13 , -15 , \ldots \}}$$

such that **Jacobi_Symbol (D,N)** is found to be equal to **-1**. To prepare for the calculation of the Lucas sequences, $\mathbf{U_n}$ and $\mathbf{V_N}$, we set the following parameters, including the parameter **R**, required by the variant algorithm:

$$\mathbf{P = 1} \text{ , } \mathbf{Q = (1-D)/4 \text{ and } R = (1+D)/2}$$
$$\mathbf{U_0 = 0} \text{ and } \mathbf{V_0 = 2}$$

Next, define the 2x2 matrix **A** which will be raised to the power **N-1** using binary modular exponentiation on matrices. This is sometimes called the Q-Matrix method and uses the right to

left binary bits of **N**, which are simple to evaluate on the fly. Matrix **T** is an auxiliary matrix which holds the result of this calculation and is initialized as the identity matrix:

$$\mathbf{A = [\ [P,-Q],[1,0]\ ]} \text{ and } \mathbf{T = [\ [1,0],[0.1]\ ]}$$

with the end result being that

$$\mathbf{T = A^{N-1}\ Mod\ N}$$

If the system you are programming this algorithm into is able to accomplish modular powers of matrices natively without complaint with very large integers, then this calculation may be performed in a fraction (~5%) of the time. At this point we calculate the two Lucas sequences, $\mathbf{U_n}$ and $\mathbf{V_n}$ with

$$\mathbf{U_{N-1} = T * [\ [P,1],[1,0]\ ]\ Mod\ N}$$
$$\mathbf{V_{N-1} = T * [\ [R,1],[1,0]\ ]\ Mod\ N}$$

which could be accomplished with simple expressions on the elements of the matrices, but I instead stuck with the matrix expressions for aesthetics since not much execution time would be saved regardless.

The items of interest are the upper left elements, $\mathbf{U_{N-1}[1,1]}$ and $\mathbf{V_{N-1}[1,1]}$. If both of the following expressions are true

$$\mathbf{U_{N-1}[1,1] = 0\ Mod\ N} \text{ and}$$
$$\mathbf{V_{N-1}[1,1] = 2*Q\ Mod\ N}$$

then **N** is extremely likely to be prime.

Results of Testing

There are no known pseudoprimes for this test. I've personally tested candidates up to 200 million with no discrepancies. Using sequence A001262 at OEIS.org, thanks to T.D. Noe, I was able to test the first 10,000 Strong Fermat Pseudoprimes to base 2 without finding a discrepancy. The last number on that list is **142,899,381,901** so this PSW variation is deterministic for any smaller candidate.

You may notice that, during the selection of parameter **D**, the loop cuts off at the 60'th attempt and then returns **"Fail"** for the test. For the average candidate, a suitable value for **D**, such that **Jacobi_Symbol (D,N)=-1**, is found by the 3'rd attempt. The problem is the outliers, that may require many many more attempts. An example would be **43,030,381**, a strong pseudoprime base 2. A suitable **D** is not found until the 35'th attempt. I've seen some take more than 40 attempts. The outliers, thankfully, are a small minority. Feel free to adjust the loop cut off to whatever you feel is appropriate. Here is a list of relatively small **N**, all strong base 2

pseudoprimes, along with the number of attempts it took to find a suitable **D**. Remember these are just the largest ones - there are many more that take ten or more attempts.

644,869 - 22
5,715,319 - 32
16,145,221 - 32
25,959,781 - 29
43,030,381 - 35

This Baillie-PSW Primality Test can handle very large integers, but be aware of performance limits if your test integer **N** is more than a couple hundred decimal digits on the HP Prime calculator. Internal issues can freeze up your calculator, requiring a reset.

Take care if you decide to experiment with the subroutine functions on their own. The function **stluc()** in particular can cause issues due to conditioning of inputs normally accomplished in the parent function **bpsw()**. The **D** parameter selection can be problematic in that it will declare **N=11** to be composite since **N=11** would normally never be an input to **stluc()** and the **D** selection method finds that 11 is divisible by 11 and thus composite!

The speed of this algorithm is impressive. As a test, try the HP Prime function **isprime()** on the 141'st Cullen Number, defined by:

$$\mathbf{C_{141} = 141*2^{141}+1 = 3930 \ldots 4833}$$

which is a known 45-digit prime. Using the function **TICKS**, we find the HP Prime determines primality in about 95 ticks, or about 95msec. The **bpsw()** test finished it's determination of $\mathbf{C_{141}}$ in just around 160 ticks in Version 1.0 and around 130 ticks in Version 1.1. Considering that **bpsw()** is a programmed function, the performance results can hardly be considered slow.

Future Work

I would like to improve this implementation by avoiding internal system hangups in the HP Prime, allowing the function to handle larger integer inputs, up to 500 decimal digits or more. The freezes I've experienced include terminal messages that suggest the use of the **simplify()** function to condition certain calculations for equality determination, etc. There is also the possibility of significant speed increases by making use of internal modular routines for powers of large integer matrices, if similar internal hangups could be avoided.

Jim Cullen
February 6, 2021