Baillie - PSW Primality Test
Variant Algorithm Pseudocode

Jim Cullen, February 2021, Ver 1.1

The Baillie–PSW primality test is a probabilistic primality testing algorithm, which this algorithm is a fast variant of. Use of the test on the HP Prime with the provided programs is straight-forward. Type in **bpsw(n)**, where **n** is the integer you wish to test for primality, and then press enter. For average integers, 20 to 100 decimal digits, the answer is returned almost instantly. **1** indicates **n** is very almost certainly prime, and **0** indicates **n** is composite. The implementation on the HP Prime is not intended for any **N** larger than about 500 decimal digits due to internal issues with modular calculations on very large integers.

The Variant PSW Test is programmed in **CAS** mode. The parent function is **bpsw()**, which in turn calls **mrpt()** and **stluc()** as subroutines. The pseudocode for **bpsw()** and both subroutines are given here for convenient translation to the programming language of your choice. The input **N** is restricted to integers greater than or equal to zero. The function returns a value of **1** to indicate true, meaning **N** is very probably prime, or returns **0** to indicate false, meaning **N** is not prime, therefore composite. If **"Fail"** is returned this means that the **stluc()** routine was not able to calculate the needed parameters and thus testing was aborted.

Comments are enclosed within **...** after each section of pseudocode to give explanation as required.

Main Routine bpsw()

```
bpsw(N)={

N=floor(abs(N));
if N<100 then
 if isprime(N) then return 1;
 else return 0; endif;
endif;

if gcd(9699690,N)>1 then return 0; endif;
if gcd(23769501540209206942181647530 3,N)>1 then return 0; endif;
```

** These lines ensure inputs are restricted to positive integers, and subsequently any value less than **100** is delegated to the internal **isprime()** function. The entire **N<100** section could be eliminated since we're only concerned with large integers, just realize that small **N** would then give erroneous results. Note that there are no local variables declared. The two gcd statements sieve out any candidates that are divisible by the first 25 primes, from 2 to 97, in two steps. The first gcd sieves prime factors 1-8 and the second gcd sieves prime factors 9-25. The large integers are the products of those primes. **

if mrpt(N,2)==0 then return 0; endif;

** This line calls the Miller-Rabin Primality Test to base 2. Subroutine **mrpt()** can be used also for other applications. **

if floor(√N)^2==N then return 0; endif;

** This line eliminates all candidates that are perfect squares. Perfect squares can not be prime and they interfere with the function of the Strong Lucas Test, so are eliminated. **

return stluc(N);
}

** If a candidate **N** survives all prior tests for compositeness, then **N** is submitted to the Strong Lucas Test, which makes the final determination for compositeness. **

Subroutine mrpt()

This subroutine performs a standard strong Fermat test, also known as the Miller-Rabin primality test. Recall that, for any prime **P**,

$$\mathbf{A^{P-1} = 1 \text{ Mod } P}$$

where **A** is the base, any integer less than **P**. What's important for the Strong Fermat Test is the power of 2 in the factors of **N-1**, which must be at least one. We simply repeat the test we did on **P-1** for **(P-1)/2**, **(P-1)/4**, **(P-1)/8**, as long as such value is a whole integer. Essentially, each is the square root of the next and, since roots of unity mod a prime can only be **1** or **-1**, then this string of Fermat tests, if arranged in a list from right to left, must appear as something similar to

$$\mathbf{\{n_1, n_2, -1, 1\}} \text{ or } \mathbf{\{1, 1, 1, 1\}} \text{ or } \mathbf{\{-1, 1, 1, 1\}}$$

In other words, a list, at least two items, of all ones, or a **-1** somewhere, followed by a string of one or more **1**'s. The final value in the list must be **1** for a prime, where $\mathbf{A^{P-1}}$ **Mod P** is evaluated. The subroutine **mrpt()** evaluates this list for any base **A**, given a candidate integer **N**.

mrpt(N,A)={
local H,D,S,P;
H=N-1; D=H; S=0;

while D mod 2 == 0
D=quotient(D/2); S=S+1;
endwhile;

** Here **D** is a working variable since we'll need **H=N-1** for later. Variable **S** holds the power of 2 in the factors of **N-1**. Variable **D** holds the value of **N-1** divided by **S** powers of 2. **

```
P=powermod(A,D,N);
if P==1 then return 1; endif;
```

** If the first result, $\mathbf{A^D}$ **Mod N = 1**, then all remaining calculations must also be **1**, so **N** is probably prime. **

```
while S>0
if P==H then return 1; endif;
P=powermod(P,2,N); S=S-1;
endwhile;
return 0;
}
```

** Calculate the remaining Fermat tests, stopping when **-1** is found, meaning all remaining Fermat tests must be a **1**, thus **N** is probably prime and we can end this portion of the test. If **-1** is not found then **N** cannot be prime, so return **0**. Recall that the variable **H** is holding the value **N-1**. **

Subroutine stluc()

This subroutine performs the Strong Lucas primality test. Any candidate **N** which passes this test, after having passed the Miller-Rabin primality test base 2, has a very high probability of being prime.

```
stluc(N)={
local B,D,H,J,K,P,Q,M,G,R,A,T,UN,VN;

B=65536; D=0; H=5;
for J=1 to 60
K=jacobi_symbol(H,N);
if K==0 then return 0; endif;
if K==-1 then D=H; break; endif;
if H>0 then H=H+2; else H=H-2; endif;
H=-H;
endfor;
if D==0 then return "Fail"; endif;
```

** This portion of the code accomplishes the task of finding a suitable **D** such that **jacobi_symbol(D,N)=-1**. The list of possible values of **D** is chosen from the list **J** of alternating positive and negative odd integers, beginning with **5**. If the result of the Jacobi Symbol check is **-1**, then **D** is set and **break** exits the for loop. If the result is **0** then we return **0** and the test ends

since **D** would divide **N**, proving **N** composite. If this code fails to find a suitable value for **D** then the test is aborted and **"Fail"** is returned. **

```
P=1; Q=(1-D)/4; M=N-1; R=(1+D)/2;
A=[ [P,-Q],[1,0] ] Mod N;
T=[ [1,0],[0,1] ] Mod N;
```

** All relevant parameters are set to perform the Lucas Sequence calculation. **R=(1+D)/2** is the one additional parameter required by this variant algorithm. **A** is a 2x2 matrix that generates the $\mathbf{U_N}$ sequence. **T** is the identity matrix that holds the result. **

```
while M>0
G:=iquorem(M,B); T=T*A^G[2] Mod N;
M=G[1];
if M==0 then break; endif;
A=A^B Mod N;
endwhile;
```

** This is a modified binary modular exponentiation on matrices that is able to process 16 bits at a time. This gives a significant speed increases since more of the calculation is performed internally by the CAS in the HP Prime. If the CAS in the HP Prime was able to handle very large integers in raising matrices to powers, then this entire section could be replaced with **T=A^M**. The Lucas Sequence is calculated and stored in matrix **T**. **

```
UN=T * [ [P,1],[1,0] ] Mod N;
if UN[1,1] != 0 then return 0; endif;
VN=T * [ [R,1],[1,0] ];
if VN[1,1] != 2*A Mod N then
   return 0;
endif;
return 1;
}
```

** The final determination of primality is performed here. The $\mathbf{U_N}$ sequence is calculated through a multiplier **[[P,1],[1,0]]** and element $\mathbf{U_N[1,1]}$ is extracted. If this is found not to equal **0 Mod N** then **N** must be composite and **0** is returned. Likewise, the $\mathbf{V_N}$ sequence is calculated through a multiplier **[[R,1],[1,0]]** and element $\mathbf{V_N[1,1]}$ is extracted. If this is found not to equal **2*Q Mod N** then **N** must be composite and **0** is returned. If all attempts to prove **N** composite by this point have failed, the algorithm relents and determines that **N** must be prime with a very high level of probability. The test returns **1** and the algorithm is terminated. **

I would be very interested to hear of any composite integer that fools this test by having itself declared prime in error. Testing all strong base 2 pseudoprimes up to the limit of $\mathbf{2^{64}}$ has yet to be accomplished and is a task for the near future.

Version Information

Version 1.0 was released Feb 7, 2021.

Version 1.1 was released Feb 11, 2021. Necessary changes to the documentation were made and new 16-bit binary modular exponentiation code was written into the **stluc()** subroutine to speed up calculation. Timing in ticks is highly variable based on the internal state of the HP Prime, even for built-in functions, but all indications are that Ver 1.1 offers a 40% reduction in execution time as compared to Ver 1.0.

Jim Cullen
February 11, 2021