Abstract

In this paper, we highlight some methods for determining if a number is a prime or a composite number. We look at the traditional brute force method and some of the probabilistic methods like Miller-Rabin, Solovay-Strassen, and Baillie PSW methods and discuss what is the best strategy when dealing with a primality testing number exceeding the limit of a 64-bit environment.

Introduction

Prime finding methods are well known. However, the question arises when you want to expand the search or just to determine if a large number (exceeding 2^{64}) is a prime number or not. We will first deal with the various methods using only 64-bit integer arithmetic and therefore only dealing with prime numbers less than 2^{64} and then expand the methods to also accommodate arbitrary precision numbers (larger than 2^{64}). The actual implementation will be a hybrid one where we use as much 64-bit arithmetic as possible and only switch to arbitrary precision when needed.

There are quite a few primes less than 2^{64}-1. The number using the prime number theorem is around $4.15 \cdot 10^{17}$.

Table of Contents

Contents

The Arbitrary precision library

If you already are familiar with the arbitrary precision library, you can skip this section. There are two classes. One for *int_precision* that handle arbitrary precision integers and one for *float_precision* that handles all *floating-point* arbitrary precision. Since Prime numbers are integers, we only need to highlight the int_precision class.

Int_precision class

To understand the C++ code and text we have to highlight a few features of the arbitrary precision library where the class name is *int_precision*. Instead of declaring, a variable with any of the build-in integer type char, short, int, long, long long, unsigned char, unsigned short, unsigned int, unsigned long, and unsigned long long you just replace the type name with *int_precision*. E.g.

```
int_precision ip;  // Declare an arbitrary precision integer
```

You can do any integer operations with *int_precision* that you can do for any type of integer in C++. Furthermore, there are a few methods you will need to know.

One of them is .iszero() which simply returns true or false if the *int_precision* variable is zero or not zero. Another is .even() and .odd() which return the Boolean value of the number even and odd status. There are other methods but I will refer you to the user manual for the arbitrary precision package [1].

Internal format for int_precision variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mNumber;
```

uintmax_t is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integer to store our integer precision number.

The method .size() returns the number of internal vector entries needed to hold the number.

The number is stored such that the vector mNumber[0] holds the least significant 64-bit binary data. The mNumber[size()-1] holds the most significant 64-bit binary data. The sign is kept separately in a class field variable mSign, which means that the mNumber holds the unsigned binary vector data.

For more details see [1].

Definition of a prime and composite number.

A prime number is a positive integer greater than 1 that has exactly two positive divisors, 1 and itself. In other words, a prime number is a number that can only be divided evenly by 1 and itself. For example, 2, 3, 5, 7, 11, and 13 are all prime numbers.

A composite number is a positive integer greater than 1 that is not a prime number. In other words, a composite number has more than two positive divisors. For example, 4, 6, 8, 9, 10, and 12 are all composite numbers and have a least one more number than one and itself that divides the number.

Example of composite numbers.

Number	**Factorization**
4	1,2,4
6	1,2,3,6
8	1,2,4,8
10	1,2,5,10
121,2	1,2,3,4,6,12

How many primes are there within a range of integers? The prime number theorem states that there are approx.:

$$no\ of\ primes \sim \frac{x}{ln(x)} \qquad (1)$$

e.g. how many primes are there in the range from 1 to 100,000. The above theorem gives 8,685 primes and the correct number is 9,592 primes. For the range of the first million the above approximation gives 72,382, and the correct number is 78,498 primes.

Brute force computation of primes

This is the classic way for the computation of primes. The algorithm is simply to test for a given number is a prime. You check if any of the preceding numbers are divisible up in the prime to check. If any number divides the prime in question, then it is a composite number and not a prime. Because there are more than two positive divisors. This is a daunting task but we can rely on a very simple optimization that reduced the amount of workload to the square root of the number. In order word for a given test of a number p is a prime. We only need to find one composite number in the range from 2 to $\sqrt{p}$. To understand why we only need to check for composite numbers up to the square root of a given number to determine if it is prime or composite, consider the following:

Suppose we have a positive integer n that we want to test for primality. If n is composite, then it can be written as a product of two positive integers a and b, where a and b are both greater than 1.

We can rewrite this as p = a · b, where a and b are factors of p. Now, at least one of a or b must be less than or equal to the square root of n, and the other factor must be greater than or equal to the square root of p. This is because if both a and b are greater than the square root of p, then their product (which is n) would be greater than n, which is a contradiction.

So, if we want to test if n is prime, we only need to check if there exists a factor of p that is less than or equal to the square root of p. If there is no such factor, then p is prime.

Therefore, we only need to check for composite numbers up to the square root of n, because any composite factor greater than the square root of n would already have been paired with a factor less than the square root of n in the above factorization.

Another optimization is that a possible prime number needs to be odd otherwise we already know that two will divide all even numbers and can therefore not be a prime. This means we will only need to check all odd numbers and can reduce the workload by 50%.

A third optimization is that all possible primes greater than 3 can be written as 6n-1 or 6n+1. This eliminates all even numbers and numbers that are divisible with 3. This means we would only need to check 5,7,11,13, 17, 19, 23, and 25,29,31, or roughly only a third of the numbers.

We can now make our first algorithm for finding prime numbers. The name prime23 indicated that we eliminate all modulo of 2 and 3 numbers to test.

Source for a simple brute force primality tester.

```
// Prime number tester
//
bool isprime23(const uintmax_t prime)
{
       uintmax_t i;

       // Handle some base cases
       if (prime <= 1)
              return false;
       if (prime == 2 || prime == 3)   // 2 & 3 is a prime
              return true;

       if (prime % 2 == 0 || prime % 3 == 0)    // Eliminate any multiple of 2 or 3.
              return false;

       // Possible prime is of the form 6*n+1 or 6*n-1
       for (i = 5; i * i <= prime; i += 6)
              if (prime % i == 0 || prime % (i + 2) == 0)
                     return false;
```

```
        return true;                                    // It is a prime
}
```

This idea can be taken one step further by eliminating all numbers divisible with 2,3 & 5. This means we only need to check 7,11,13, 17, 19, 23, and 29 for the first 30 numbers and then the cycle (for the next 30 numbers) repeats itself as 37,41,43,47,49,53 and 59. This will further reduce the number to check by another 6%.

The name for the function isprime235 indicates that we have eliminated all modulo of the first 10 primes and then only test numbers that are not a modulo of 2, 3, and 5. This means more numbers to skip and reduce the workload by 8-9% compare to the first simpler version.

Source for a better brute force primality tester

```
// Prime number tester
//
bool isprime235(const uintmax_t prime)
{
        uintmax_t i;

        // This table handle reduce the check of the next 30 numbers to only 8
        // instead of 10 in the isprime23 version
        //Eliminating any multiple of 2,3 & 5.
        // 10 first known primes
        const uintmax_t precheck[11] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
        const uintmax_t primes[8] = {  1, 7, 11, 13, 17, 19, 23, 29 };
        uintmax_t kp;

        // Handle some base cases
        if (prime <= 1)
                return false;

        for (i = 0; i < 10; ++i)
                if ((prime % precheck[i]) == 0)
                        return prime == precheck[i];

        // kp<sqrt(prime) becomes kp*kp<primes
        // Loop to only test 8 numbers in a group of 30 numbers per loop
        for (kp=30; kp * kp < prime; kp += 30)
        {
                for (i = 0; i < 8; ++i)
                {
                        if (prime % (kp + primes[i]) == 0)
                                return false;           //meaning its not prime
                }
        }

        return true;                                    // It is a prime
}
```

There is certainly some limitation to using the brute force method, which is slow performance when testing of a prime number exceeds 10^8-10^{10}. When the need calls for the higher prime number to test then you would need to switch to the Probabilistic Primality testers to gain an acceptable performance.

Probabilistic Primality Testers

There are several methods in this category of probabilistic primality testers. Among them are:

- Solovay-Strassen
- Miller-Rabin
- Baillie PSW
- Lucas-Lehmer
- And others

Se [4].

Comparing Primality tester methods

The Miller-Rabin primality test is a probabilistic algorithm, meaning that it can produce false positives (and false negatives) with some probability. A false positive number is a composite number that is declared as a prime and consequently, a false negative is a prime that is declared as a composite number. However, the algorithm is highly efficient and can determine the primality of a number with high accuracy, especially when using a sufficient number of rounds (iterations) of the test.

Compared to other primality tests, the Miller-Rabin test is generally considered to be faster than most deterministic tests for large numbers. Deterministic tests, such as the AKS test, can determine the primality of a number with absolute certainty, but they are generally less efficient than probabilistic tests like the Miller-Rabin test and therefore not used in practice.

The Miller-Rabin primality test is a reliable and widely-used algorithm that mathematicians and computer scientists have studied and analyzed in depth. It's also pretty easy to implement, which makes it very practical for applications in cryptography and number theory.

In practice, the Miller-Rabin test has proven to be highly effective at determining whether a number is prime or not. It's considered to be one of the best probabilistic primality tests available since it's both efficient and accurate.

That being said, there are other primality tests out there too. For example, the Solovay-Strassen test is another probabilistic test that's quite similar to the Miller-Rabin test, but it uses a different approach based on the Euler criterion.

The Solovay-Strassen test is generally considered to be slightly slower than the Miller-Rabin test, but it is also more accurate in some cases.

Baillie-PSW test: This is a probabilistic primality test that combines elements of the Lucas-Lehmer test with the Miller-Rabin test. The Baillie-PSW test is also considered to be one of the most accurate probabilistic tests available.

Lucas-Lehmer test: This is a deterministic primality test that is specifically designed to test for Mersenne primes (numbers of form 2^p - 1) and is most likely the fastest deterministic test available for large Mersenne primes.

Elliptic curve primality test (ECPP): This is a deterministic primality test that is based on elliptic curves. The ECPP test is faster than most deterministic tests, but it is also more complex to implement.

Just like anything else, each of these primality tests has its own strengths and weaknesses. When it comes to choosing which one to use, it depends on the specific situation and what you're looking for in terms of speed and accuracy.

A workable source code version in C++ for most of these methods can easily be found by browsing the web. E.g. [7]. These versions usually are implemented using either 32-bit or 64-bit arithmetic however this can be extended to arbitrary precision fairly easily. Usually, you would implement a hybrid solution where you test if the number to be tested is within the reach of 64-bit arithmetic and then use the faster native version and only switch to arbitrary precision when needed. Now there are some issues you would need to overcome using 64-bit arithmetic and that is how to handle the arithmetic overflow.

Handling arithmetic overflow

Many of the probabilistic testers required many computations of the form: a^b modulo c or a·b modulo c. Here we have a risk of arithmetic overflow when first computing the a^b or a·b. Luckily there is a way to calculate the result without overflowing in the interim steps.

The product of a·b can be handled by the following algorithm using repeated squaring.

Avoiding overflow comes at the price of more operations. While a·b modulo requires 2 arithmetic operations the algorithm above requires between 3-5 per loop and depending on the value of b up to 63 operations as a worst case on a 64-bit system.

```
result = 0;
while (b)
  if (b & 1)
      result = (result + a) % c
  a = (a * 2) % c
  b >>= 1
return result
```

*Algorithm 1. (a*b)%c*

In order words care much be taken to only use the above algorithm when there is a risk of overflowing an arithmetic operation.

The C++ source for imul_mod()

```cpp
// Perform (a*b)%c without causing overflow in the interim multiplication
intmax_t imul_mod(intmax_t a, intmax_t b, const intmax_t c)
{
	intmax_t result = 0;
	for (; b > 0; b >>= 1)
	{
		if (b & 1)
			result = (result + a) % c;
```

```
                a = (a * 2) % c;
        }
        return result;
}
```

This support function is only needed when using 64-bit arithmetic. When using arbitrary precision arithmetic there is no overflow to prevent. (As the name applied)

The same goes for a^b modulo c where a similar algorithm as the above can be used to avoid overflowing.

```
result=1
a = a % c
While(b>0)
    if (b & 1)
        result = (result * a )%c [1)]
    a =( a * a)%c [1)]
    b>>=1
return result
```

Algorithm 2. (a^b)%c

Notice [1)] the multiplication can overflow so you would need to use algorithm 1 for the multiplication and modulo.
Here the operations per loop are also between 3 and 5 and the loop can go from 1 to 63.

The C++ source for ipow_modulo()

```
// Modular exponentiation function (a^b mod c)
// Computation is done to prevent overflow by using imul_mod for critical
// multiplication of a*b
// If multiplication can not be done safely then we use imul_mod()
intmax_t ipow_modulo(intmax_t a, intmax_t b, intmax_t c)
{
        intmax_t res = 1;
        intmax_t p = a;

        // multiplication can be done safely then do a*b
        // otherwise do imul_mod(a,b,c)
        for (; b > 0; b >>= 1)
        {
                if (b & 0x1)
                {
                        if(p<INT_MAX && res<INT_MAX)
                                res = (res * p) % c;
                        else
                                res = imul_mod(res, p, c);
                }
                if (p < INT_MAX)
                        p = (p * p) % c;
                else
                        p = imul_mod(p,p,c);
                if (res < 0)
                        res = res;
        }
        return res;
}
```

ipow_modulo() already exists in the arbitrary precision packages [1] and can be used directly. Although not strictly needed in arbitrary precision it can however speed up the calculation since we are dealing with smaller arbitrary precision numbers in the interim steps and particularly using the modulo operations with a 'smaller' argument can increase the performance.

Jacobi Symbol.

Another prerequisite for using some of these testers (e.g. Baillie PSW and Solovay-Strassen) is the availability to compute the Jacobi symbol.

The Jacobi symbol is a mathematical function with a lot of practical applications in number theory and cryptography. It's an extension of the Legendre symbol, which helps us check if a number is a quadratic residue modulo a prime.

But the Jacobi symbol goes beyond that: it helps us determine if a number is a quadratic residue modulo any odd composite number, not just primes. This is why it's used in important cryptographic algorithms like RSA and ElGamal encryption.

Beyond cryptography, the Jacobi symbol has important applications in other areas of mathematics, such as algebraic number theory, elliptic curves, and Galois theory. It's even useful in some algorithms for solving diophantine equations and factoring large integers.

Overall, the Jacobi symbol is a powerful mathematical tool that's widely used in both theoretical and practical math, as well as in cryptography.

A function to compute the Jacobi symbol can be found on https://rosettacode.org/wiki/Jacobi_symbol#C++ and with some modification to allow for negative arguments, you can use the below function.

Source code for the Jacobi function.

```
// Calculate the Jacobian symbol for a given number (a/n)
//The result is either -1,0 or 1
int jacobi(intmax_t a, intmax_t n)
{
	int result = 1;
	int n_mod_8;

	if (a < 0)
	{	// use quadratic reciprocity law
		a = -a;
		if (n % 4 == 3)
			result = -result;
	}

	a %= n;

	while (a != 0)
	{
		while (a % 2 == 0)
		{
			a >>= 1;
			n_mod_8 = n % 8;
```

```
                        if (n_mod_8 == 3 || n_mod_8 == 5)
                                result = -result;
                }
                swap(a, n);
                if (a % 4 == 3 && n % 4 == 3)
                        result = -result;
                a %= n;
        }
        if (n == 1)
                return result;
        else
                return 0;
}
```

.

Jacobi in arbitrary precision

It is straightforward to alter the above Jacobi function to handle arbitrary precision.

The arbitrary precision version of the Jacobi function.

```
// Calculate the Jacobian symbol for a given number (a/n)
// This is the arbitrary precision version
// The result is either -1,0 or 1
int_precision jacobi(const int_precision& a, const int_precision& n)
{
        const int_precision c2(2), c4(4), c8(8);
        int_precision aip(a), nip(n);
        int result = 1; // can only be -1,0, or 1
        int n_mod_8;

        // Check if we can use the 64-bit version of Jacobi
        if( a.size()==1&&n.size()==1&&((a.index(0) & (1ull << 63)) == 0) &&
            ((n.index(0) & (1ull<<63))==0))
                return jacobi(intmax_t(a),intmax_t(n));

        if (aip.sign() < 0)
        {       // use quadratic reciprocity law
                aip.change_sign();
                if (int(nip % c4) == 3)
                        result = -result;
        }

        aip %= nip;

        while (!aip.iszero())  // While not zero
        {
                while (aip.even())
                {
                        aip >>= 1;
                        n_mod_8 = int(nip % c8);
                        if (n_mod_8 == 3 || n_mod_8 == 5)
                                result = -result;
                }
                swap(aip, nip);
                if (int(aip % c4) == 3 && int(nip % c4) == 3)
                        result = -result;
                aip %= nip;
        }
        if (nip == int_precision(1))
```

```
		return int_precision(result);
	else
		return int_precision(0);
}
```

The above function is a so-called hybrid implementation that automatically detects if the computation can be carried out using 64-bit arithmetic which delivers a performance gain when the parameter a and n is within the 64-bit environment.

Solovay-Strassen

The Solovay-Strassen primality test is a type of probabilistic primality test that was first introduced by Robert M. Solovay and Volker Strassen back in 1977. This clever test is based on the Euler criterion, which is a special case of Euler's criterion for determining whether a number is a quadratic residue modulo a prime.

Like other probabilistic primality tests, the Solovay-Strassen test uses randomness to figure out whether a number is likely to be prime or composite. The test works by selecting a random number "a" and computing the Jacobi symbol of "a" and "n", where "n" is the number being tested for primality. If the Jacobi symbol is equal to the modular exponentiation of "a" modulo "n" using Euler's criterion, then it's likely that "n" is a prime number.

The Solovay-Strassen test is pretty easy to use and is often employed as a first-pass test before more sophisticated ones like the Miller-Rabin test or the Baillie-PSW test. However, it's not perfect and can sometimes generate false positives or false negatives. For this reason, the Solovay-Strassen test is usually combined with other tests to increase the reliability of the primality test.

Despite its limitations, the Solovay-Strassen test has played an important role in the development of primality testing algorithms. It has helped pave the way for more sophisticated probabilistic and deterministic tests and has contributed to our understanding of number theory and cryptography. The tester is described in more detail in [6].

Accuracy of the Solovay-Strassen test

The accuracy per iteration is only half of the Miller-Rabin (see later). This means that if we required the same accuracy for both the Miller-Rabin test and the Solovay-Strassen test we need to perform twice as many iterations to keep the probability the same between the two test methods. For the Solovay-Strassen the probability, p for a correct answer is:

$$(1/2)^k = p \Rightarrow \qquad (2)$$

$$k = \log(1/p) / \log(2) \qquad (3)$$

At first glance this seems not very accurate however in practical tests you usually get much higher accuracy. E.g. Scanning the first 10M positive integers reveal only approx. 385 false results. Yielding an error rate of 0.00057 instead of the probability of a correct answer to be 0.5 this is 877 times less than the expected rate of 0.5. One extra iteration reduced the number of false results to 19 or an error rate of 2.8e-05 or ~ 8,900 less than the expected rate of 0.25. With 5 iterations the error rate was down to approx. 0.

First 664,579 primes in the range of the first 10,000,00 integers				
Rounds/Iterations	False Positive	False Negative	Error rate	Probabilistic error rate
1	385	0	5.79e-4	0.5
2	19	0	2.86e-5	0.25
3	7	0	1.05e-5	0.125
4	1	0	1.50e-6	0.063
5	0	0	0	0.031
6	1	0	1.50e-6	0.016
7	0	0	0	0.008

As expected, the more rounds (iteration) the fewer false positives. Also, note that the accuracy result is many times better than the probabilistic error bound for Solovay-Strassen.

64-bit source code for the Solovay-Strassen primality tester. The original was from the Solovay-Strassen method of Primality Test - GeeksforGeeks, however, modified by using a more advanced random generator with a uniform distribution.

```
// Perform the Solovay-Strassen Primality Test
static bool SolovayStrassen(const uintmax_t p, const int iterations)
{
	// Check for small primes
	if (p <= 1) return false;
	if (p == 2 || p == 3) return true;
	// Eliminate any multiple of 2 or 3.
	if (p % 2 == 0 || p % 3 == 0) return false;

	// Instead of just using rand() this is a much stronger way to generate random
	// numbers in the range 2,p-2, and up to 64bit environment
	// Although it slower the performance quite significantly, it is more randomness
	random_device rd;
	mt19937_64 generator(rd());
	uniform_int_distribution<intmax_t> dis(2, p - 2);

	for (int i = 0; i < iterations; i++)
	{
		// Generate a random number a
		uintmax_t a = dis(generator);
		uintmax_t jacobian = (p + jacobi(a, p)) % p;
		uintmax_t mod = ipow_modulo(a, (p - 1) / 2, p);

		if (!jacobian || mod != jacobian)
			return false;
	}
	return true;
}
```

Arbitrary precision Solovay-Strassen

We don’t have, at least not directly access to an arbitrary precision version of the above random generator, at least not of the same level as the build in the C++ language standard library. Instead, we use a workaround. We know that an arbitrary precision integer internally consists of several 64-bit unsigned vector elements. [1]. We can get access to how many 64-bit elements comprise the arbitrary precision number by using the method .size(). We then randomly decide how many 64-bit elements our random number needs to have. A number between 1 and the size() of the arbitrary precision number. And then we build up the random number by calling the random generator several times to build up the entire random number for our arbitrary precision number. Other than that, it follows the 64-bit version.

Source code for the arbitrary precision version of the Solovay-Strassen primality tester.

```
// To perform the Solovay-Strassen Primality Test using arbitrary precision
static bool SolovayStrassen(const int_precision& p, const int iterations)
{       const int_precision c1(1), c2(2), c3(3);
        int_precision aip;
        uintmax_t a;

        // Check for small primes
        if (p <= c1) return false;
        if (p == c2 || p == c3) return true;
        // Eliminate any multiple of 2 or 3.
        if ((p % c2).iszero() || (p % c3).iszero()) return false;

        // Instead of just using rand() this is a much stronger way to generate random
        // numbers in the range 2,p-2, and up to 64bit environment
        // Although it slower the performance quite significantly, it is more randomness
        random_device rd;
        mt19937_64 generator(rd());
        a = p.index(p.size() - 1);
        uniform_int_distribution<uintmax_t> disindex(1, p.size());
        uniform_int_distribution<uintmax_t> dishigh(2, a - 2);
        uniform_int_distribution<uintmax_t> dis(2, (~0ull) - 2);

        for (int i = 0; i < iterations; i++)
        {
                // Generate a random number aip
                a = dishigh(generator);
                uintmax_t inx = 0;

                if (p.size() > 1)  // dont call disindex is p.size()==1 or max 64-bit data
                        inx = disindex(generator);

                aip = a;  // Set most significant
                for (; inx > 1; --inx)
                {
                        aip <<= Bitsiptype;
                        aip += dis(generator);
                }

                int_precision jacobian = (p + jacobi(aip, p)) % p;
                int_precision mod = ipow_modulo(aip, (p - c1) / c2, p);

                if (jacobian.iszero() || mod != jacobian)
                        return false;
        }
        return true;
}
```

Miller-Rabin

The Miller-Rabin primality test is a popular method for determining whether a given number is prime or composite, using a probabilistic algorithm. It's known as a "probabilistic" test because instead of giving a definite answer, it provides a likely answer that's usually very accurate but not always guaranteed to be correct.
To understand the test, it's important to know about Fermat's Little Theorem, which states that if p is a prime number and a is any integer not divisible by p, then a^{p-1} is congruent to 1 modulo p. In simpler terms, if we take any number a and raise it to the power p-1 (where p is a prime number), the result will always be 1 when divided by p.

The Miller-Rabin test takes advantage of this fact by testing a given number n with a set of randomly generated values of a. If n is a prime number, then a^{n-1} will always be congruent to 1 modulo n, for any value of a that is not divisible by n. If n is composite, then there will be at least some values of a for which a^{n-1} is not congruent to 1 modulo n.

The algorithm first writes n-1 as 2s · r, where r is an odd number. This is possible because any even number can be factored into powers of 2, and any odd number can be expressed as 2k+1 for some integer k. Then, for each randomly generated value of a, the algorithm computes a^r mod n. If this value is 1 or n-1, then n is likely to be prime. If not, the algorithm computes $a^{(2^\wedge j \cdot r)}$ mod n for j from 1 to s-1. If any of these values are equal to n-1, then n is likely to be prime. If none of them are, then n is composite.

The test is repeated for a certain number of randomly generated values of a. The more values of a that are tested, the higher the likelihood that the algorithm correctly identifies whether n is prime or not. However, it's important to note that there's always a small chance of the algorithm producing a false positive, which means identifying a composite number as prime. The chance can be reduced by increasing the number of tests performed.

Overall, the Miller-Rabin primality test is a widely-used tool in cryptography and other fields where large prime numbers need to be generated or verified.

The Invention of the Miller-Rabin Tester

The Miller-Rabin primality test was developed by Michael O. Rabin and Gary L. Miller, with Miller publishing the algorithm in 1976 and Rabin improving it in 1980. Before the Miller-Rabin test, the Lucas-Lehmer test was commonly used for primality testing, but it was only effective for a specific type of prime numbers known as Mersenne primes.

The Miller-Rabin primality test was a significant breakthrough in primality testing, as it provides a fast and accurate way to determine whether a given number is prime or composite. The algorithm uses probabilistic methods to quickly identify composite numbers, while still being highly accurate for identifying prime numbers. This has made the Miller-Rabin primality test widely used in various fields, such as cryptography, computer science, and number theory.

Although Rabin and Miller were both credited with jointly inventing the Miller-Rabin primality test, their contributions to the development of the algorithm were different. Miller introduced a probabilistic primality test in 1976 that used a single random base to test whether a given number is prime or composite. His algorithm was based on Fermat's Little Theorem, which states that if p is a prime number and a is any positive integer not divisible by p, then $a^{(p-1)} \equiv 1 \pmod{p}$. Miller's algorithm tested whether a given number n is prime by randomly selecting a base a and then checking if $a^{(n-1)} \equiv 1 \pmod{n}$. If this congruence was not satisfied, then n was composite. If it was satisfied, then n may be prime, but further tests were needed.

In 1980, Rabin improved Miller's algorithm by introducing a variant that used multiple random bases to increase the accuracy of the test. Rabin's algorithm tested whether a given number n is prime by selecting k random bases a_1, a_2, ..., a_k, and then checking if each of the congruences $a\_i^{(n-1)} \equiv 1 \pmod{n}$ was satisfied. If any of the congruences were not satisfied, then n was composite. If all of the congruences were satisfied, then n was considered a probable prime. Rabin showed that the probability of a false positive result could be made arbitrarily small by choosing a suitable value of k.

In summary, the Miller-Rabin primality test is a significant development in the field of primality testing, with its impact felt in various fields today. Miller's contribution was the initial probabilistic primality test based on Fermat's Little Theorem, while Rabin's contribution was the improvement of the algorithm by using multiple random bases to increase the accuracy of the test.

The accuracy of the Miller-Rabin primality test

The accuracy of the Miller-Rabin primality test increases with the number of iterations or rounds of the test. In general, the more rounds of the test are performed, the lower the probability of a false positive result, i.e., the probability of identifying a composite number as prime.

Specifically, for each iteration of the test, the probability of a false positive result is at most 1/4, meaning that the test is correct at least 75% of the time. So, for example, if we perform the test with 10 randomly chosen bases, the probability of a false positive result is at most $(1/4)^{10} \sim 9.5 \cdot 10^{-7}$, which is less than one in a million.

There is a formula that can be used to calculate the number of iterations of the test required to achieve the desired level of accuracy. Let p be the probability of a false positive result, and let k be the number of iterations of the test. Then, the probability of at least one false positive result after k iterations are at most $(1/4)^k$, so we can set this equal to p and solve for k:

$$(1/4)^k = p \Rightarrow \tag{4}$$

$$k = \log(1/p) / \log(4) \tag{5}$$

For example, if we want the probability of a false positive result to be less than one in a billion ($p=10^{-9}$), we can calculate that we need k = 15 iterations of the test.

It's important to note that the Miller-Rabin primality test is a probabilistic algorithm, so it can still produce false positives (and false negatives) even with a large number of iterations. However, for practical purposes, the number of iterations can be chosen to provide a level of accuracy that is sufficient for the intended application. E.g. in cryptography, you will typically use 64-128 iterations to get an acceptable result. The Miller-Rabin primality tester is described in more detail in [5].

First 664,579 primes in the range of the first 10,000,00 integers				
Rounds/Iterations	False Positive	False Negative	Error rate	Probabilistic error rate
1	211	0	3.17e-4	0.25
2	19	0	1.66e-5	0.063
3	11	0	3.01e-6	0.016
4	2	0	1.50e-6	0.063
5	1	0	0	3.906e-3

As expected, we see the same pattern as in the Solovay-Strassen method where the more rounds (iteration) the fewer false positives. Also, note that the accuracy result is many times better than the probabilistic error bound for Miller-Rabin.

Source code for the Miller-Rabin primality test. We used the same way to generate more advanced random numbers to be used for the Miller-Rabin test as we did in the Solovay-Strassen test.

```
// Miller-Rabin primality test
// Works for up to 64bit integers
//
static bool MillerRabin(const intmax_t n, int k)
{
	// Handle some base cases
	if (n == 2 || n == 3)
		return true;
	if (n <= 1 || n % 2 == 0 || n % 3 == 0) // Eliminate factors of 2 and 3
		return false;

	// Find r and d such that n-1 = 2^r * d
	int r;
	intmax_t d = n - 1;
	for (r = 0; d % 2 == 0; ++r)
		d >>= 1;

	// Perform k iterations of the Miller-Rabin test
	random_device rd;
	mt19937_64 gen(rd());
	uniform_int_distribution<intmax_t> dis(2, n - 2);

	for (int i = 0; i < k; i++)
	{
		intmax_t a = dis(gen);
		intmax_t x = ipow_modulo(a, d, n);
		if (x == 1 || x == n - 1)
			continue;

		for (int j = 0; j < r - 1; j++)
		{
			x = ipow_modulo(x, 2, n);
			if (x == n - 1)
				break;
		}
```

```
			if (x != n - 1)
				return false;
		}
		return true;
}
```

Arbitrary precision Miller-Rabin

We use the same trick as before for generating an arbitrary precision number.

```
// Miller-Rabin primality test
// Works for arbitrary precision integers
//
static bool MillerRabin(int_precision& n, const int k)
{
	const int_precision c1(1), c2(2), c3(3);
	int_precision aip;
	uintmax_t a, r;


	// Handle some base cases
	if (n == c2 || n == c3)
		return true;
	if (n <= c1 || n.even())
		return false;

	// Find r and d such that n-1 = 2^r * d
	r = 0;
	// find r, d so that d is odd and (2^r)d = n-1
	int_precision d = n - c1;
	r = d.ctz();  // Faster than the while loop. while (d.even()) {d >>= 1; r++;}
	d >>= r;

	// Perform k iterations of the Miller-Rabin test
	random_device rd;
	mt19937_64 generator(rd());  // Initialized random generator with random seed

	a = n.index(n.size()-1);
	uniform_int_distribution<uintmax_t> disindex(1,n.size());
	uniform_int_distribution<uintmax_t> dishigh(2, a-2);
	uniform_int_distribution<uintmax_t> dis(2, (~0ull) - 2);

	for (int i = 0; i < k; i++)
	{
		a = dishigh(generator);
		uintmax_t inx=0;

		if(n.size()>1)  // dont call disindex is n.size()==1 or max 64-bit data
			inx = disindex(generator);

		aip = a;  // Set most significant
		for (; inx > 1; --inx)
		{
			aip <<= Bitsiptype;
			aip += dis(generator);
		}

		int_precision x = ipow_modulo(aip, d, n);
```

```
                if (x == c1 || x == n - 1)
                        continue;

                for (int j = 0; j < r - 1; j++)
                {
                        x = ipow_modulo(x, c2, n);
                        if (x == n - 1)
                                break;
                }
                if (x != n - c1)
                        return false;
        }

        return true;
}
```

Baillie PSW

The Baillie PSW primality test was created by Colin Baillie and Robert S. Wagoner in the early 1980s. The test is named after the initials of their surnames and is also known as the Baillie-PSW test. It is a probabilistic primality test that combines the strengths of the Miller-Rabin test and the Lucas test. The Miller-Rabin test is a primality test based on Fermat's Little Theorem, while the Lucas test is a specialized primality test based on Lucas sequences, which arise in number theory.

Baillie and Wagoner developed the Baillie PSW test as a way to create a more reliable and efficient primality test. The test is popular among mathematicians and computer scientists and is widely used in cryptographic applications. One of the important features of the Baillie PSW test is its use of strong pseudoprimes, a type of composite number that can be difficult to distinguish from prime numbers and passes certain primality tests.

Baillie and Wagoner showed that the Baillie PSW test is reliable even for strong pseudoprimes, making it a useful tool for testing large numbers for primality. The Baillie PSW test has contributed to the development of other probabilistic and deterministic primality tests used in various applications.

However, the author notes that some implementations of the Baillie PSW test may not be accurate and that some tests may not be true Baillie PSW tests. One implementation that the author cites claims to implement a strong Lucas test, which is a specialized primality test that is not the same as the Baillie PSW test. The author believes that this implementation works correctly without false positives or negatives for the first 2^{64} -1 numbers.

The Baillie PSW primality tester is described in more detail in [3]. And is considerably more complicated to implement than the two previous methods.

.Algorithm 3. The Baillie PSW primality test

1. Check n for some based prime numbers of 2 and 3 and any modulo hereof.

2. Perform some trial division for primes up to 101. This is done to ensure that in Step 4 we can make a reasonable choice of D.
3. Perform a base 2 single round of the Miller-Rabin test. If n is not a strong probable prime then n must be composite and return false.
4. Find the first D in the sequence 5, −7, 9, −11, 13, −15, … etc. for which the Jacobi symbol (D/n) is −1. Set P = 1 and Q = (1 − D) / 4.
5. Perform a normal or strong Lucas probable prime test on n using parameters D, P, and Q. If n is not a strong Lucas probable prime, then n is composite. Otherwise, n is almost certainly prime.

Accuracy of the Baillie PSW primality test.

Testing the first 10M numbers for prime using the Baillie PSW test you get zero false positives and zero false negatives and that is without using the strong Lucas properly prime test. This makes the Baillie PSW a very accurate test. The author of the test claims that it works without errors for any number below 2^{64}-1.

Now is there a way to estimate the probability of the test giving you the correct answer? To my knowledge, there does not exist a proper answer for this other than the error rate is very low. This can make the use of the Baillie PSW test in some situations difficult to use since you have no way of knowing how much you can trust the result. You don't have that problem with the Solovay-Strassen or the Miller-Rabin test since there is a well-known formula for the probability of a correct answer.

The source was derived from a Python version that used a mix of functions from other sources. The code has been converted and optimized for use in C++ and rewritten of code to reduce arithmetic overflow in a 64-bit environment. Furthermore, some of the sub-function has been implemented as C++ Lambda function. The only outside function is the Lucas strong probability test.

```
// Function to compute the Baillie-PSW test
bool BailliePSW(const uintmax_t n, const bool spp=false)
{
	// Check for small primes
	if (n <= 1) return false;
	if (n == 2 || n == 3) return true;
	if (n % 2 == 0 || n % 3 == 0) return false;// Eliminate any multiple of 2 or 3.

	// need to test small primes as the D chooser might not find
	// a suitable value for small primes
	for (uintmax_t p: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
				59, 61, 67, 71, 73, 79, 83, 89, 97, 101})
		if (n % p == 0)
			return n == p;

	// Check for perfect squares
	uintmax_t sqrt_n = uintmax_t(sqrt(n));
	if (sqrt_n * sqrt_n == n)
		return false;
	// This implementation uses a single base, 2, but is written in a way
	// that allows additional bases to be added easily if required
	// The last line, commented out, is an example of how the routine could be
	// modified.
	// if more bases are required.
	// It is implemented as a Lambda function
	auto miller_rabin = [](uintmax_t a, uintmax_t n)
	{
		// find s, d so that d is odd and (2^s)d = n-1
		uintmax_t d = n - 1, s = 0;
		while ((d & 1) == 0) {
			d >>= 1;
			s++;
		}
		// test whether 'n' is a strong probable prime to base 'a'
		a = ipow_modulo(a, d, n);
		if (a == 1)
			return true;
		for (int r = 0; r < s - 1; r++)
		{
			if (a == n - 1)
				return true;
			a = ipow_modulo(a, 2, n);
		}
		return (a == n - 1);
	};

	// Do one Miller Rabin step with base ==2
	if (miller_rabin(2,n) == false)  // if (miller_rabin(2, d, n, s) &&
miller_rabin(3, d, n, s) && miller_rabin(5, d, n, s));
		return false;   // Composite number

	// Find Dj pair
	pair<int, int> Dj;
	//Dj = D_chooser(n);  // D_chooser do not need to check for is_square()
	Dj.first = 5;
	Dj.second = jacobi(Dj.first, n);
	while (Dj.second > 0)
	{
		Dj.first += (Dj.first > 0 ? 2 : -2);
		Dj.first *= -1;
		Dj.second = jacobi(Dj.first, n);
	}
	if (Dj.second == 0)
		return false;

	// Check for normal or strong Lucas
	return lucas_spp(n,Dj.first, 1, (1-Dj.first)/4, spp);
}
```

And the supporting lucas_spp() function. Notice the fifth parameter that indicates if we use a strong or extra strong Lucas test.

```
// Lucas strong probable prime test
static bool lucas_spp(uintmax_t n, intmax_t D, intmax_t P, intmax_t Q, bool spp)
{
	pair<intmax_t, intmax_t> UV;

	if (n % 2 == 0) return false;

	// Lambda function for (a*b)%n without overflow
	auto mod_mul = [](intmax_t a, intmax_t b, const intmax_t n)
	{
		intmax_t result = 0;
		for (; b > 0; b >>= 1)
		{
			if (b & 1)
				result = (result + a) % n;
			a = (a * 2) % n;
		}
		return result;
	};

	// Lamnda function to Compute U_k and V_k in the Lucas sequence
	auto UVseq = [](const intmax_t k, const intmax_t n, const intmax_t P, const 
intmax_t D) {
		intmax_t U = 1, V = P;
		bitset<64> dig(k);
		int i;

		// Lambda function for dividing with 2 modulo n
		// Assumes n is odd
		auto div2modn = [](const intmax_t a, const intmax_t n)
		{
			intmax_t x = a;
			if (x & 1)
				x += n;
			return (x >> 1) % n;
		};

		// Lambda function for detection of multiplication overflow
		auto overflow = [](const intmax_t a, const intmax_t b)
		{
			if (a != 0 && (b >= INT64_MAX / a || b < INT64_MIN / a))
				return true;
			return false;
		};

		// Lambda function for (a*b)%n without overflow
		auto mod_mul = [](intmax_t a, intmax_t b, const intmax_t n)
		{
			intmax_t result = 0;
			for (; b > 0; b >>= 1)
			{
				if (b & 1)
					result = (result + a) % n;
				a = (a * 2) % n;
			}
			return result;
		};

		// Bypass all leading zeros
		for (i = int(dig.size()) - 1; i >= 0 && 0 == dig[i]; --i)
			;
		// Bypass the first one bit
		--i;
		// Take each of the remaining bits
		for (; i >= 0; --i)
		{
			intmax_t tmp = U;

			U = mod_mul(U, V, n);		// (U*V)%n
```

```
                    if (overflow(V, V) || overflow(D * tmp, tmp)) // 64bit overflow?
                    {
                           intmax_t Vtmp = V, oldV = V, oldU = tmp, oldtmp = tmp;
                           Vtmp = mod_mul(Vtmp, Vtmp, n);
                           tmp = mod_mul(tmp, tmp, n);
                           tmp = mod_mul(D, tmp, n);
                           Vtmp += tmp;
                           V = div2modn(Vtmp, n);
                    }
                    else
                    {
                           V = div2modn(V * V + D * tmp * tmp, n);  // V'2k=(
(V'k)^2+D(U'k)^2)/2
                    }
                    while (V < 0) //ensures that V is positive modulo n.
                           V += n; // Emulates floored division
                    if (1 == dig[i])
                    {// Notice P and D are usually small numbers so we don't check for
overflow here
                           tmp = U;
                           U = div2modn(P * U + V, n);       // U'k+1=(P(U'k)+(V'k))/2
                           V = div2modn(D * tmp + P * V, n); // V'k+1=(D(U'k)+P(V'k))/2
                           while (V < 0)
                                  V += n;// Ensures that V is positive modulo n.
Emulates floored division
                    }
             }
             pair<intmax_t, intmax_t> UV(U, V);
             return UV;
      };

      // Normal Lucas test
      if (spp == false)
      {      // Regular Lucas
             UV = UVseq(n + 1, n, P, D);
             return (UV.first == 0);
      }

      uintmax_t d = n + 1, s;
      for (s = 0; d % 2 == 0; ++s)
             d >>= 1;

      intmax_t U, V;

      UV=UVseq(d, n, P, D);
      U=UV.first; V = UV.second;
      if (U == 0 || V == 0)
             return true;

      // Compute the strong Lucas
      Q = ipow_modulo(Q, d, n);
      while (Q < 0)
             Q += n;             // Ensured correct flored division

      // Lambda function for detection of multiplication overflow
      /*auto overflow = [](const intmax_t a, const intmax_t b)
      {
             if (a != 0 && (b >= INT64_MAX / a || b < INT64_MIN / a))
                    return true;
             return false;
```

```
	};
	*/

	for (; s>0; --s)
	{
		// V = V * V - 2 * Q;
		int64_t v_mod_n = V % n, q_mod_n = Q % n, vv_mod_n = mod_mul(v_mod_n, 
v_mod_n, n);
		int64_t two_q_mod_n = (2 * q_mod_n) % n;
		V = (vv_mod_n - two_q_mod_n + n) % n;

		if (V >= 0)
			V %= n;
		else
		{	// We need a flored division, not a truncated division as in C++
			while (V < 0)
				V += n;
			//	V %= n; not needed
		}
		if (V == 0)
			return true;
		// Double the subscript. Q square is always positive so the c++ % operator
		// is safe to use
		Q = ipow_modulo(Q, 2, n); // (Q * Q) % n;
	}
	return false;
}
```

Arbitrary precision Baillie PSW

As with the other primality tester, it is fairly easy to convert it into an arbitrary version.

```
// Function to compute the Baillie-PSW primality test
bool BailliePSW(const int_precision& n, const bool spp=false)
{
	const int_precision c1(1), c2(2), c3(3);

	// Check for small primes
	if (n <= c1) return false;
	if (n == c2 || n == c3) return true;
	// Eliminate any multiple of 2 or 3.
	if ((n % c2).iszero() || (n % c3).iszero()) return false;

	// Need to test small primes as the D chooser might not find
	// a suitable value for small primes
	for (uintmax_t p: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 
				59, 61, 67, 71, 73, 79, 83, 89, 97, 101})
		if ((n % int_precision(p)).iszero())
			return n == int_precision(p);

	// Check for perfect squares
	auto is_square = [](const int_precision& n)
	{
		const int_precision c1(1);
		if (n.sign() < 0) return false;
		if (n.iszero()) return true;

		int_precision x(1), y(n);
		while (x + c1 < y)
		{
			int_precision mid = (x + y) >> c1;
			if (mid * mid < n)
				x = mid;
			else
				y = mid;
		}
		return (n == x * x || n == (x + c1) * (x + c1));
	};

	// If perfect square then it is a composite number
	if (is_square(n))
		return false;

	// This implementation uses a single base, 2, but is written in a way
	// that allows additional bases to be added easily if required
	// The last line, commented out, is an example of how the routine could be
	// modified.
	// if more bases are required.
	// It is implemented as a Lambda function
	auto miller_rabin = [](const int_precision& a, const int_precision& n)
	{
		const int_precision c1(1), c2(2);
		int_precision aip(a);
		// find s, d so that d is odd and (2^s)d = n-1
		int_precision d = n - c1;
		intmax_t s;
		s = d.ctz();  // Faster than the while loop. while (d.even()){d>>=1; r++;}
		d >>= s;

		// test whether 'n' is a strong probable prime to base 'a'
		aip = ipow_modulo(aip, d, n);
		if (aip == c1)
			return true;
		for (intmax_t r = 0; r < s - 1; r++)
		{
			if (aip == n - c1)
				return true;
			aip = ipow_modulo(aip, c2, n);
		}
		return (aip == n - c1);
	};

	// Do one Miller Rabin step with base ==2
	if (miller_rabin(c2, n) == false)  // if (miller_rabin(2, d, n, s) &&
miller_rabin(3, d, n, s) && miller_rabin(5, d, n, s));
		return false;   // Composite number

	// Find Dj pair
	pair<int, int> Dj;
	//Dj = D_chooser(n);
	Dj.first = 5;
	Dj.second = int(jacobi(int_precision(Dj.first), n));

	while (Dj.second > 0)
	{
		Dj.first += (Dj.first > 0 ? 2 : -2);
		Dj.first *= -1;
		Dj.second = int(jacobi(int_precision(Dj.first), n));
	}
	if (Dj.second == 0)
```

```
            return false;

        // Check for normal or strong Lucas
        return lucas_spp(n, Dj.first, 1, (1 - Dj.first) / 4, spp);
}
```

And the supporting lucas_spp() function for arbitrary precision. Notice the fifth parameter that indicates if we use a strong or extra strong Lucas test.

```
// Lucas strong probable prime test for arbitrary precision
static bool lucas_spp(const int_precision& n, const intmax_t D, const intmax_t P, const
intmax_t QQ, bool spp)
{
        const int_precision c1(1), c2(2);
        pair<int_precision, int_precision> UV;

        if (n.even()) return false;

        // Lambda function to Compute U_k and V_k in the Lucas sequence
        // This is the arbitrary precision version
        auto UVseq=[](const int_precision & k, const int_precision & n, const intmax_t P,
const intmax_t D) {
                int_precision U = int_precision(1), V = P;
                intmax_t i, j;

                // Lambda function for dividing with 2 modulo n
                // Assumes n is odd
                auto div2modn = [](const int_precision& a, const int_precision& n)
                {
                        int_precision x(a);
                        if (x.odd())
                                x += n;
                        return (x >> 1) % n;
                };

                // Take each of the bits in the int_precisioon number
                for (j = k.size(); j > 0; --j)
                {
                        bitset<64> dig(k.index(j - 1));
                        if (j == k.size())  // First time
                        {// Bypass all leading zeros in the most significant vector
                                for (i = intmax_t(dig.size()) - 1; i >= 0 && 0 == dig[i]; --
i)
                                        ;
                                // Bypass the first one bit in the most significant vector
                                --i;
                        }
                        else
                                i = 63; // Otherwise take all the bits from the subsequent
vector
                        for (; i >= 0; --i)
                        {
                                int_precision tmp = U;

                                U = (U * V) % n;
                                V = div2modn(V * V + D * tmp * tmp, n);  // V'2k=(
(V'k)^2+D(U'k)^2)/2

                                while (V < 0)
                                        V += n;// ensure that V is positive modulo n. Emulates
floored division
```

```
                            if (1 == dig[i])
                            {       // Notice P and D are usually small numbers so we
don't check for overflow here
                                    tmp = U;
                                    U = div2modn(P * U + V, n);// U'k+1=(P(U'k)+(V'k))/2
                                    V = div2modn(D * tmp + P * V, n);
       //V'k+1=(D(U'k)+P(V'k))/2
                                    while (V < 0)
                                           V += n;// Ensure that V is positive modulo n.
Emulates floored division
                            }
                     }
              }
              pair<int_precision, int_precision> UV(U, V);
              return UV;
       };


       // Normal Lucas test
       if (spp == false)
       {
              UV = UVseq(n + c1, n, P, D);
              return (UV.first.iszero() );
       }
       // Strong Lucas
       int_precision d = n + c1;
       uintmax_t s;
       for (s = 0; d.even(); ++s)
              d >>= c1;

       int_precision U, V;

       UV = UVseq(d, n, P, D);
       U = UV.first; V = UV.second;
       if (U.iszero() || V.iszero())
              return true;

       // Compute the strong Lucas
       int_precision Q(QQ);
       Q = ipow_modulo(Q, d, n);
       while (Q < 0)
              Q += n;                // Ensured correct floored division

       for (; s>0; --s)
       {
              V = V * V - c2 * Q;
              if (V >= 0)
                     V %= n;
              else
              {      //We need a flored division, not a truncated division as in C++
                     while (V < 0)
                            V += n;
                     //     V %= n; not needed
              }
              if (V.iszero())
                     return true;
              // Double the subscript. Q square is always positive so the c++ % operator
is safe to use
              Q = (Q * Q) % n;
       }
       return false;
```

```
}
```

Performance of Primality Testers

For prime numbers exceeding 2^{64} Baillie PSW is a clear winner. Verifying primes for 20-300 digits the Baillie PSW at least twice as fast as the Miller-Rabin test. For prime numbers less than 2^{64} they were both performance-wise in the same neighborhood whereas the Solovay-Strassen test was half the speed of the Miller-Rabin.

Recommendation for primality Tester

Since the performance of the Solovay-Strassen for the equivalent iterations reaching a certain probabilistic level lacks Miller-Rabin there is no need to consider that method over Miller-Rabin. If we compare the Miller-Rabin with the Baillie PSW our performance test show that the Baillie PSW is significantly faster than the Miller-Rabin test for prime number exceeding 20 decimal digits. The Baillie PSW should be the chosen one. However, it is very hard to bind the probabilistic level of the Baillie PSW test and although it is very accurate it is a probabilistic primality tester and if you need some assurance that the probabilistic result is below a certain threshold you would need to consider the Miller-Rabin tester. In [1] both methods have been implemented.

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages
2) ChatGPT (www.openai.com) on March 5, 2023
3) Wikipedia Baillie PSW primality tester. Baillie–PSW primality test - Wikipedia
4) Wikipedia Primality tester. Primality test - Wikipedia
5) Wikipedia Miller-Rabin primality tester. Miller–Rabin primality test - Wikipedia
6) Wikipedia Solovay-Strassen primality tester. Solovay–Strassen primality test - Wikipedia
7) Geeks for geeks. Primality Test | Set 3 (Miller–Rabin) - GeeksforGeeks