The Miller-Rabin primality test is a simple probabilistic algorithm for determining whether a number is prime or composite that is easy to implement. It proves compositeness of a number using the following formulas:

Suppose $0 < a < n$ is coprime to $n$ (this is easy to test using the GCD). Write the number $n-1$ as $2^s \cdot d$, where $d$ is odd. Then, provided that all of the following formulas hold, $n$ is composite:

$$a^d \not\equiv 1 \pmod{n}$$
$$a^{2^r d} \not\equiv -1 \pmod{n} \text{ for all } 0 \leq r \leq s - 1$$

If $a$ is chosen uniformly at random and $n$ is prime, these formulas hold with probability $1/4$. Thus, repeating the test for $k$ random choices of $a$ gives a probability of $1 - 1/4^k$ that the number is prime. Moreover, Gerhard Jaeschke showed that any 32-bit number can be deterministically tested for primality by trying only $a=2$, 7, and 61.

32-bit integers

We begin with a simple implementation for 32-bit integers, which is easier to implement for reasons that will become apparent.
32-bit modular exponentiation function

```java
private static int modular_exponent_32(int base, int power, int modulus) {
    long result = 1;
    for (int i = 31; i >= 0; i--)
        if ((power & (1 << i)) != 0)
            result = (result * base) % modulus;
    return (int)result; // Will not truncate since modulus is an int
}
```

Int is a 32-bit integer type and long is a 64-bit integer type. Since the product of two 32-bit numbers has at most 64 bits, we know for certain that this will never overflow or truncate bits.

Next, we write a function that actually tests the desired formulas for a given a and n. It works by starting with \(a^d\) and squaring it \(s-1\) times, comparing each to \(-1\) (\(n-1\) actually; we have to add \(n\) because we're working with nonnegative numbers):

It returns true for prime and false for composite.

32-bit Miller-Rabin pass

```java
private static boolean miller_rabin_pass_32(int a, int n) {
    int a_to_power = modular_exponent_32(a, d, n);
    if (a_to_power == 1) return true;
    for (int i = 0; i < s-1; i++)
        if (a_to_power == n-1) return true;
    a_to_power = modular_exponent_32(a_to_power, 2, n);
    if (a_to_power == n-1) return true;
    return false;
}
```

The call to `modular_exponent_32` to square `a_to_power` could be converted into a call to a more specialized function, but we omit this optimization here. To compute \(s\) and \(d\), we just divide \(n-1\) by 2 until it becomes odd:

32-bit compute s and d

```java
int d = n - 1; int s = Integer.numberOfTrailingZeros(d); d >>= s;
```

Now, taking advantage of Jaeschke's result, we can write a deterministic, 100% accurate implementation that tries the values of 2, 7, and 61 for \(a\), returning true if they all produce true, and false otherwise:
<<32-bit Miller-Rabin>>= public static boolean miller_rabin_32(int n) { if (n <= 1) return false; else if (n == 2) return true; else if (miller_rabin_pass_32(2, n) && (n <= 7 || miller_rabin_pass_32(7, n)) && (n <= 61 || miller_rabin_pass_32(61, n))) return true; else return false; }

The tests at the beginning are special cases, since Miller-Rabin requires \( n \geq 3 \). The other tests on \( n \) ensure we don't pass an \( a \) that greater than or equal to \( n \). Here's some test code that uses it:

<<MillerRabin32.java>>= public class MillerRabin32 {
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    System.out.println(miller_rabin_32(n) ? "PRIME" : "COMPOSITE"));
  }
}

|^edit|

Arbitrary-precision integers

The test for arbitrary-precision integers is largely identical except that arbitrary-precision integer operations should be used. We will use Java's BigInteger class.

We then rewrite our implementation above using these functions. We don't need a modular exponent function because the BigInteger class provides us with a \( \text{modPow}(\) \) method.

When performing the Miller-Rabin pass, only \( a, n, d \), and powers of \( a \) are large numbers; \( s \) and \( i \) are logarithmically smaller. We precompute the large integers 1 and \( n-1 \) for use in comparisons and squaring:
<<Miller-Rabin pass>>= private static boolean miller_rabin_pass(BigInteger a, BigInteger n) {
    BigInteger n_minus_one = n.subtract(BigInteger.ONE);
    BigInteger a_to_power = a.modPow(d, n);
    if (a_to_power.equals(BigInteger.ONE)) return true;
    for (int i = 0; i < s-1; i++) {
        if (a_to_power.equals(n_minus_one)) return true;
        a_to_power = a_to_power.multiply(a_to_power).mod(n);
    }
    if (a_to_power.equals(n_minus_one)) return true;
    return false;
}

We’ve replaced the squaring using `modular_exponent` with a modular multiply of `a_to_power` by itself. Finally, we use bit shifts to compute `d` rapidly, and test just its least significant component to determine if it is odd:

<<compute s and d>>= BigInteger d = n_minus_one; int s = d.getLowestSetBit(); d = d.shiftRight(s);

Since our numbers are now arbitrarily large, we can no longer take advantage of Jaeschke’s result to produce a deterministic implementation, but we can produce a highly accurate probabilistic implementation by simply running the test enough times to make \(1/4^k\) very small, say 20 times.

<<Miller-Rabin>>= public static boolean miller_rabin(BigInteger n) {
    for (int repeat = 0;
        repeat < 20; repeat++) {
        BigInteger a;
        do { a = new BigInteger(n.bitLength(), rnd); }
        while (a.equals(BigInteger.ZERO));
        if (!miller_rabin_pass(a, n)) {
            return false;
        }
    } return true;
}

For convenience we ignore the special cases where \(n \leq 2\).

Finally, some test code:
We've augmented `main()` with the ability to generate a prime number of a specified number of bits. It does this by randomly selecting numbers of that size until it finds a prime one. We could use this to rapidly generate large primes for use in cryptography. Since most random values have small prime factors, we first test some of these to avoid an expensive test:

```java
if (p.mod(BigInteger.valueOf(2)).equals(BigInteger.ZERO)) continue;
if (p.mod(BigInteger.valueOf(3)).equals(BigInteger.ZERO)) continue;
if (p.mod(BigInteger.valueOf(5)).equals(BigInteger.ZERO)) continue;
if (p.mod(BigInteger.valueOf(7)).equals(BigInteger.ZERO)) continue;
```

Here's some sample output:

```bash
$ java MillerRabin test 516119616549881 PRIME $ java MillerRabin test 516119616549887 COMPOSITE
$ java MillerRabin genprime 128 277003545840181465186202237665148044033 $ java MillerRabin
genprime 512 690706701483912052342329574373473828883687815325200193110546590937028225649711\62269141121965617545613897865904269544243721348093953041840506282278098321
```

Download code