

Probable prime – Botan

```
/*
 * Generate a random prime
 */
BigInt random_prime(RandomNumberGenerator& rng,
                    size_t bits, const BigInt& coprime,
                    size_t equiv, size_t modulo)
{
    if(bits <= 1)
        throw Invalid_Argument("random_prime: Can't make a prime of " +
                                to_string(bits) + " bits");

    else if(bits == 2)
        return ((rng.next_byte() % 2) ? 2 : 3);
    else if(bits == 3)
        return ((rng.next_byte() % 2) ? 5 : 7);
    else if(bits == 4)
        return ((rng.next_byte() % 2) ? 11 : 13);

    if(coprime <= 0)
        throw Invalid_Argument("random_prime: coprime must be > 0");
    if(modulo % 2 == 1 || modulo == 0)
        throw Invalid_Argument("random_prime: Invalid modulo value");
    if(equiv >= modulo || equiv % 2 == 0)
        throw Invalid_Argument("random_prime: equiv must be < modulo, and odd");

    while(true)
    {
        BigInt p(rng, bits);

        // Force lowest and two top bits on
        p.set_bit(bits - 1);
        p.set_bit(bits - 2);
        p.set_bit(0);

        if(p % modulo != equiv)
            p += (modulo - p % modulo) + equiv;

        const size_t sieve_size = std::min(bits / 2, PRIME_TABLE_SIZE);
        SecureVector<size_t> sieve(sieve_size);

        for(size_t j = 0; j != sieve.size(); ++j)
            sieve[j] = p % PRIMES[j];

        size_t counter = 0;
        while(true)
        {
            if(counter == 4096 || p.bits() > bits)
                break;

            bool passes_sieve = true;
            ++counter;
            p += modulo;

            if(p.bits() > bits)
                break;

            for(size_t j = 0; j != sieve.size(); ++j)
            {
                sieve[j] = (sieve[j] + modulo) % PRIMES[j];
                if(sieve[j] == 0)
                    passes_sieve = false;
            }

            if(!passes_sieve || gcd(p - 1, coprime) != 1)
                continue;
            if(check_prime(p, rng))
                return p;
        }
    }
}

/*
 * Test for primality using Miller-Rabin
 */
bool primality_test(const BigInt& n,
                    RandomNumberGenerator& rng,
                    size_t level)
{
    {
        const size_t PREF_NONCE_BITS = 128;

        if(n == 2)
            return true;
        if(n <= 1 || n.is_even())
            return false;

        // Fast path testing for small numbers (<= 65521)
        if(n <= PRIMES[PRIME_TABLE_SIZE-1])
        {
            const word num = n.word_at(0);

            for(size_t i = 0; PRIMES[i]; ++i)
            {
                if(num == PRIMES[i])
                    return true;
                if(num < PRIMES[i])
                    return false;
            }

            return false;
        }

        if(level > 2)
            level = 2;

        const size_t NONCE_BITS = std::min(n.bits() - 2, PREF_NONCE_BITS);

        MillerRabin_Test mr(n);

        if(mr.is_witness(2))
            return false;

        const size_t tests = miller_rabin_test_iterations(n.bits(), level);

        for(size_t i = 0; i != tests; ++i)
        {
            BigInt nonce;
            while(nonce < 2 || nonce >= (n-1))
                nonce.randomize(rng, NONCE_BITS);

            if(mr.is_witness(nonce))
                return false;
        }

        return true;
    }
}
```