

Probable prime – Crypto++

```
bool Integer::GenerateRandomNoThrow(RandomNumberGenerator &i_rng, const
NameValuePairs &params)
{
    Integer min = params.GetValueWithDefault("Min", Integer::Zero());
    Integer max;
    if (!params.GetValue("Max", max))
    {
        int bitLength;
        if (params.GetIntValue("BitLength", bitLength))
            max = Integer::Power2(bitLength);
        else
            throw InvalidArgument("Integer: missing Max argument");
    }
    if (min > max)
        throw InvalidArgument("Integer: Min must be no greater than Max");

    Integer equiv = params.GetValueWithDefault("EquivalentTo", Integer::Zero());
    Integer mod = params.GetValueWithDefault("Mod", Integer::One());

    if (equiv.IsNegative() || equiv >= mod)
        throw InvalidArgument("Integer: invalid EquivalentTo and/or Mod
argument");

    Integer::RandomNumberType rnType =
params.GetValueWithDefault("RandomNumberType", Integer::ANY);

    member_ptr<KDF2_RNG> kdf2Rng;
    ConstByteArrayParameter seed;
    if (params.GetValue(Name::Seed(), seed))
    {
        ByteQueue bq;
        DERSequenceEncoder seq(bq);
        min.DEREncode(seq);
        max.DEREncode(seq);
        equiv.DEREncode(seq);
        mod.DEREncode(seq);
        DEREncodeUnsigned(seq, rnType);
        DEREncodeOctetString(seq, seed.begin(), seed.size());
        seq.MessageEnd();

        SecByteBlock finalSeed((size_t)bq.MaxRetrievable());
        bq.Get(finalSeed, finalSeed.size());
        kdf2Rng.reset(new KDF2_RNG(finalSeed.begin(), finalSeed.size()));
    }
    RandomNumberGenerator &rng = kdf2Rng.get() ? (RandomNumberGenerator
&)*kdf2Rng : i_rng;

    switch (rnType)
    {
        case ANY:
            if (mod == One())
                Randomize(rng, min, max);
            else
            {
                Integer min1 = min + (equiv-min)%mod;
                if (max < min1)
                    return false;
                Randomize(rng, Zero(), (max - min1) / mod);
                *this *= mod;
                *this += min1;
            }
            return true;

        case PRIME:
        {
            const PrimeSelector *pSelector =
params.GetValueWithDefault(Name::PointerToPrimeSelector(), (const PrimeSelector
*)NULLPTR);

            int i;
            i = 0;
            while (1)
            {
                if (++i==16)
                {
                    // check if there are any suitable primes in
[min, max]

                    Integer first = min;
                    if ( (first, max, equiv, mod, pSelector))
                    {
                        // if there is only one suitable
prime, we're done

                        *this = first;
                        if (!FirstPrime(first, max, equiv,
mod, pSelector))

                            return true;
                    }
                    else
                        return false;
                }

                Randomize(rng, min, max);
                if (FirstPrime(*this,
STDMIN(*this+mod*PrimeSearchInterval(max), max), equiv, mod, pSelector))
                    return true;
            }

            default:
                throw InvalidArgument("Integer: invalid RandomNumberType
argument");
        }
    }
}

bool FirstPrime(Integer &p, const Integer &max, const Integer &equiv, const Integer
&mod, const PrimeSelector *pSelector)
{
    CRYPTOPP_ASSERT(!equiv.IsNegative() && equiv < mod);

    Integer gcd = GCD(equiv, mod);
    if (gcd != Integer::One())
    {
        // the only possible prime p such that p%mod==equiv where
GCD(mod,equiv)!=1 is GCD(mod,equiv)
        if (p <= gcd && gcd <= max && IsPrime(gcd) && (!pSelector ||
pSelector->IsAcceptable(gcd)))
        {
            p = gcd;
            return true;
        }
        else
            return false;
    }

    unsigned int primeTableSize;
    const word16 * primeTable = GetPrimeTable(primeTableSize);

    if (p <= primeTable[primeTableSize-1])
    {
        const word16 *pItr;

        --p;
        if (p.IsPositive())
            pItr = std::upper_bound(primeTable,
primeTable+primeTableSize, (word)p.ConvertToLong());
        else
            pItr = primeTable;

        while (pItr < primeTable+primeTableSize && !(*pItr%mod == equiv && (!
pSelector || pSelector->IsAcceptable(*pItr)))
            ++pItr;

        if (pItr < primeTable+primeTableSize)
        {
            p = *pItr;
            return p <= max;
        }

        p = primeTable[primeTableSize-1]+1;
    }

    CRYPTOPP_ASSERT(p > primeTable[primeTableSize-1]);

    if (mod.IsOdd())
        return FirstPrime(p, max, CRT(equiv, mod, 1, 2, 1), mod<<1,
pSelector);

    p += (equiv-p)%mod;

    if (p>max)
        return false;

    PrimeSieve sieve(p, max, mod);

    while (sieve.NextCandidate(p))
    {
        if ((!pSelector || pSelector->IsAcceptable(p)) &&
FastProbablePrimeTest(p) && IsPrime(p))
            return true;
    }

    return false;
}
```