

Intro To Asymmetric Cryptography

RPISEC

Avi Weinstock (aweinstock)

October 29, 2019

```
static_prime(BIGNUM *rnd, int bits) {
    int i;
    uint32_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        if (bits >= BN_BITS2) {
            /* if bits is so small that it fits into a single word then we
             * additionally don't want to exceed that many bits. */
            if (is_single_word) {
                BN_ULONG size_limit;
                if (bits == BN_BITS2) {
                    /* avoid undefined behavior. */
                    size_limit = (((BN_ULONG)0) - get_word(rnd));
                } else {
                    size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
                }
                if (size_limit < maxdelta) {
                    delta = size_limit;
                } else {
                    delta = 0;
                }
            }
        } else {
            delta = 0;
        }

loop:
        if (is_single_word) {
            BN_ULONG rwd_word = get_word(rnd);

            /* In the case that the candidate prime is a single word then
             * we check that:
             * 1) It's greater than primes[i] because we shouldn't reject
             *    3 as being a prime number because it's a multiple of
             *    three.
             * 2) That it's not a multiple of a known prime. We don't
             *    check that rwd-1 is also coprime to all the known
             *    primes because there aren't many small primes where
             *    that's true. */
            for (i = 1; i < NUMPRIMES && primes[i] < rwd_word; i++) {
                if ((mods[i] + delta) % primes[i] == 0) {
                    delta += 2;
                    if (delta > maxdelta) {
                        goto again;
                    }
                    goto loop;
                }
            }
        } else {
            for (i = 1; i < NUMPRIMES; i++) {
                /* check that rnd is not a prime and also
                 * that get(rnd-1,primes) == 1 (except for 2) */
                if ((BN_mod_word(rnd-1, primes[i]) <= 1) &&
                    (i > 1 || BN_mod_word(rnd, primes[i]) <= 1)) {
                    delta += 2;
                    if (delta > maxdelta) {
                        goto again;
                    }
                }
            }
        }
    }
}
```

Symmetric vs Asymmetric Cryptosystems

Examples of symmetric crypto: Examples of asymmetric crypto:

- ▶ classical ciphers (caesar, vigenere)
- ▶ block ciphers (DES, AES)
- ▶ stream ciphers (OTP, RC4, Salsa20)
- ▶ hash functions (MD5, SHA256)
- ▶ MACs (HMAC, Poly1305)

- ▶ key exchange (Diffie-Hellman)
- ▶ encryption (RSA, ElGamal)
- ▶ signatures (RSA, DSA, Schnorr)
- ▶ homomorphic encryption (Paillier, RSA, Gentry)

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    int n = mods[NUMPRIMES];
    int delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a number 'rnd' to test. */
    BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
    if (mod == (BN_ULONG)-1) {
        return 0;
    }
    mods[i] = (uint16_t)mod;

    /* small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit = BN_BITS2;
        /* avoid underflow below 0. */
        size_limit = (((BN_ULONG)0) - get_word(rnd));
    } else {
        size_limit = ((BN_ULONG)0) << bits - get_word(rnd) - 1;
    }
    if (size_limit < maxdelta) {
        maxdelta = size_limit;
    }
    delta = 2;
loop:
    if (is_single_word) {
        if (BN_mod_word(rnd, mod) == 0) {
            /* In the case that the candidate prime is a single word then
             * we check that:
             * 1) It's greater than primes[i] because we shouldn't reject
             * 3 as being a prime number because it's a multiple of
             * three.
             * 2) That it's not a multiple of a known prime. We don't
             * check that rnd-1 is also coprime to all the known
             * primes because there aren't many small primes where
             * that's true. */
            for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
                if ((mods[i] + delta) % primes[i] == 0) {
                    delta += 2;
                    if (delta > maxdelta) {
                        goto again;
                    }
                    goto loop;
                }
            }
        } else {
            for (i = 1; i < NUMPRIMES; i++) {
                /* check that rnd is not a prime and also
                 * that gcd(rnd-1, primes) == 1 (except for 2) */
                if ((mods[i] + delta) % primes[i] <= 1) {
                    delta += 2;
                }
            }
        }
    }
}
```

What is the significance of RSA?

- ▶ Historically the first asymmetric cryptosystem (published in 1977)
- ▶ Named for its inventors: Rivest, Shamir, Adleman
- ▶ Very flexible: can be used for both encryption and signing, is multiplicatively homomorphic
- ▶ Easy to implement
- ▶ Easy to mess up implementing, on account of its flexibility

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }

    /* If 'rnd' is a single word then we
    additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            delta = size_limit;
        } else {
            delta = 0;
        }
    }

loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
        we check that:
        1) That it's not a multiple of 2, because we shouldn't reject
        2) That it's not a multiple of a known prime. We don't
        check that rnd-1 is also coprime to all the known
        primes because there aren't many small primes where
        that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
            * that gcd(rnd-1,primes) == 1 (except for 2) */
            if ((mods[i] + delta) % primes[i] <= 1) {
                delta += 2;
            }
        }
    }
}
```

What is RSA?

- ▶ p and q are distinct large primes
- ▶ $n = p * q$
- ▶ $\varphi(n) = (p - 1) * (q - 1)$
- ▶ $e * d \equiv 1 \pmod{\varphi(n)}$
- ▶ (n, e) is the "public key"
- ▶ (p, q, d) is the "private key"
- ▶ $\text{enc}(x) = \text{rem}(x^e, n)$
- ▶ $\text{dec}(x) = \text{rem}(x^d, n)$

```
static prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    else is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 0;

loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         * 3 as being a prime number because it's a multiple of
         * three.
         * 2) That it's not a multiple of a known prime. We don't
         * check that rnd-1 is also coprime to all the known
         * primes because there aren't many small primes where
         * that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    }
    else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if ((mods[i] + delta) % primes[i] <= 1) {
                delta += 2;
            }
        }
    }
}
```

Euclidean division

$$\begin{array}{r}
 27 \leftarrow q \\
 3 \overline{) 82} \leftarrow x \\
 \underline{6} \\
 22 \\
 \underline{21} \\
 1 \leftarrow r
 \end{array}$$

$y = 3$
 $x = 82$
 $r = 1$

$$\forall x, y \exists q, r (y * q + r = x)$$

$$q = \text{quot}(x, y)$$

$$r = \text{rem}(x, y)$$

$$0 \leq r < y$$

$$3 * 27 + 1 = 82$$

```

static prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits < BN_BITS2) {
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
        delta = 0;
    }
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);
        /* we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         * 3 as being a prime number because it's a multiple of
         * three.
         * 2) That it's not a multiple of a known prime. We don't
         * check that rnd-1 is also coprime to all the known
         * primes because there aren't many small primes where
         * that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that get(rnd-1,primes) = 1 (except for 2) */
            if ((mods[i] + delta) % primes[i] <= 1) {
                delta += 2;
            }
        }
    }
}

```

Modular congruence and primes

Divides cleanly/Is divisor of:

- ▶ $\text{divides}(x, y) \leftrightarrow \text{rem}(y, x) = 0$
- ▶ e.g. $\text{divides}(5, 10)$, since $10 \% 5 == 0$

Modular congruence:

- ▶ $x \equiv y \pmod{n} \leftrightarrow \text{divides}(x - y, n)$
- ▶ $x \equiv y \pmod{n} \leftrightarrow \text{rem}(x, n) = \text{rem}(y, n)$

Primeness:

- ▶ $\text{prime}(p) \leftrightarrow \forall k \in [2, p-1](\neg \text{divides}(k, p))$
- ▶

```
def prime(p):  
    return all([p % k != 0 for k in range(2,p)])
```

`assert filter(prime, range(2, 20)) == [2, 3, 5, 7, 11, 13, 17, 19]`

Greatest Common Divisor:

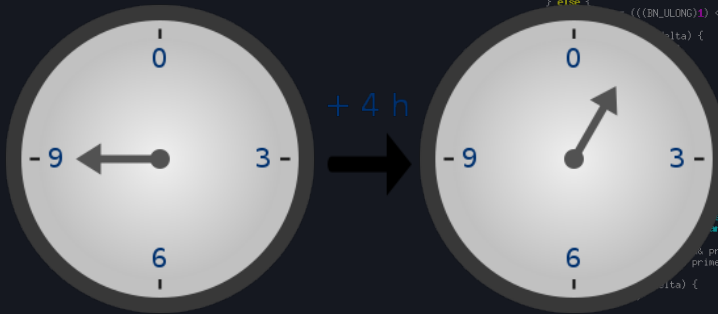
- ▶ $\text{gcd}(x, y) = \max\{k | \text{divides}(k, x) \wedge \text{divides}(k, y)\}$
- ▶

```
def gcd(x, y):  
    return max([1]+[k for k in range(1, x*y) if x % k == 0 and y % k == 0])
```

```
static int probable_prime(BIGNUM *rnd, int bits) {  
    int i;  
    uint16_t mods[NUMPRIMES];  
    BN_ULONG delta;  
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];  
    char is_single_word = bits <= BN_BITS2;  
  
    again:  
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) < 0)  
        return 0;  
  
    /* we now have a random number 'rnd' to test. */  
    for (i = 1; i < NUMPRIMES; i++) {  
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);  
        if (mod == (BN_ULONG)-1) {  
            return 0;  
        }  
        mods[i] = (uint16_t)mod;  
    }  
    /* If bits is so small that it fits into a single word then we  
     * additionally don't want to exceed that many bits. */  
    if (is_single_word) {  
        BN_ULONG size_limit;  
        if (bits == BN_BITS2) {  
            /* Avoid undefined behavior. */  
            size_limit = (((BN_ULONG)0) - get_word(rnd));  
        } else {  
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;  
        }  
        if (size_limit < maxdelta) {  
            maxdelta = size_limit;  
        }  
        delta = 0;  
    }  
    /* If it's a single word */  
    BN_ULONG rnd_word = get_word(rnd);  
  
    /* In the case that the candidate prime is a single word then  
     * we check that:  
     * 1) It's greater than primes[i] because we shouldn't reject  
     * 3 as being a prime number because it's a multiple of  
     * 2) that it's not a multiple of a known prime. We don't  
     * check that rnd-1 is also coprime to all the known  
     * primes because there aren't many small primes where  
     * that's true. */  
    for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {  
        if ((mods[i] + delta) % primes[i] == 0) {  
            delta += 2;  
            goto loop;  
        }  
        if (i == NUMPRIMES - 1) {  
            /* Check that rnd-1 is coprime to all the known  
             * primes because there aren't many small primes where  
             * that's true. */  
            if ((rnd_word - 1) % primes[i] == 0) {  
                delta += 2;  
                goto loop;  
            }  
        }  
    }  
    return 1;  
}
```

Modular arithmetic

- ▶ Let \mathbb{Z}_n denote $\{\text{rem}(x, n) \mid x \in \mathbb{Z}\}$, or equivalently $[0, n)$.
- ▶ We can truncate addition and multiplication to work within \mathbb{Z}_n by calculating remainders after each operation.
- ▶ "Clock arithmetic": in \mathbb{Z}_{12} , $9 + 4 = 1$, since $\text{rem}(9 + 4, 12) = \text{rem}(13, 12) = 1$



```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    BN_ULONG mod = BN_mod_l(rnd, 0, BN_ULONG)primes[i];
    if (mod == ((BN_ULONG)-1)) {
        return 0;
    }
    mods[i] = (uint16_t)mod;

    /* it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        /* avoid undefined behavior. */
        size_limit = "((BN_ULONG)0) - get_word(rnd);
    } else {
        size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
    }
    BN_ULONG delta = (size_limit / mods[i]) % mods[i];

    /* The prime is a single word then
     * [i] because we shouldn't reject
     * or because it's a multiple of
     * one of a known prime. We don't
     * also coprime to all the known
     * weren't many small primes where
     * & primes[i] < rnd_word; i++) {
     *   primes[i] == 0) {
     *       delta += 2;
     *   }
     * }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1,primes) = 1 (except for 2) */
            if ((mod = rnd->word[0] + primes[i]) <= 1)
                delta += 2;
        }
    }
    goto loop;
}

return 1;
}
```

Addition and Multiplication mod 7

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

*	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

```

static prime(BIGNUM *rnd, int bits) {
    int i;
    uint32_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint32_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
       additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit =
            (bits == BN_BITS2) ?
                size_limit = (((BN_ULONG)0) - get_word(rnd));
            else
                size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 1;
loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);
    }

    /* In the case that the candidate prime is a single word then
       we check that:
       * 1) It's greater than primes[i] because we shouldn't reject
       * 3 as being a prime number because it's a multiple of
       * three.
       * 2) That it's not a multiple of a known prime. We don't
       * check that rnd-1 is also coprime to all the known
       * primes because there aren't many small primes where
       * that's true. */
    for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
        if ((mods[i] + delta) % primes[i] == 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
} else {
    for (i = 1; i < NUMPRIMES; i++) {
        /* check that rnd is not a prime and also
           * that get(rnd-1,primes) == 1 (except for 2) */
        if ((mods[i] + delta) % primes[i] == 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
}

```


Addition and Multiplication mod 8

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

*	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

```

static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint32_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == ((BN_ULONG)-1)) {
            return 0;
        }
        mods[i] = (uint32_t)mod;

        if (bits is small, but it fits into a single word then we
        additionally can't want to check that mod is bits. */
        if (is_single_word) {
            BN_ULONG size_limit;
            if (bits == BN_BITS2) {
                /* avoid undefined behavior. */
                size_limit = (((BN_ULONG)0) - get_word(rnd));
            } else {
                size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
            }
            if (size_limit < maxdelta) {
                maxdelta = size_limit;
            }
        }
        if (is_single_word) {
            BN_ULONG rnd_word = get_word(rnd);
            /* In the case that the candidate prime is a single word then
            * we check that:
            * 1) It's greater than primes[i] because we shouldn't reject
            * 3 as being a prime number because it's a multiple of
            * three.
            * 2) That it's not a multiple of a known prime. We don't
            * check that rnd-1 is also coprime to all the known
            * primes because there aren't many small primes where
            * that's true. */
            for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
                if ((mods[i] + delta) % primes[i] == 0) {
                    delta += 2;
                    if (delta > maxdelta) {
                        goto again;
                    }
                    goto loop;
                }
            }
        } else {
            for (i = 1; i < NUMPRIMES; i++) {
                /* check that rnd is not a prime and also
                * that gcd(rnd-1, primes) == 1 (except for 2) */
                if ((mods[i] + delta) % primes[i] != 1) {
                    delta += 2;
                    if (delta > maxdelta) {
                        goto again;
                    }
                }
            }
        }
    }
}

```

Multiplicative inverses

- ▶ In \mathbb{Q} and \mathbb{R} , inverses exist everywhere except zero: $x * \frac{1}{x} = 1$
- ▶ In \mathbb{Z} , inverses only exist at 1
- ▶ In \mathbb{Z}_p , inverses exist everywhere except zero and can be found via fermat's little theorem (e.g. $3 * 4 \equiv 12 \equiv 1 \pmod{11}$)
- ▶ In \mathbb{Z}_n , multiples of factors of n act as additional zeros for the purposes of not having an inverse; when they exist, they can be found via an extension of the algorithm for euclidean division

```
static prime(BIGNUM *rnd, int bits) {
    int i;
    uint32_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (!is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* avoid overflowed behavior */
            size_limit = ((BN_ULONG)1) << (bits - 1);
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
        delta = 0;
    }
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);
        /* If the candidate prime is a single word then
         * we check that:
         * 1) It's not a multiple of 2 (we should reject
         *    being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) == 1 (except for 2) */
            if ((BN_ULONG)rnd_word % primes[i] <= 1) {
                delta += 2;
            }
        }
    }
}
```

Fermat's little theorem

- ▶ $x^{p-1} \equiv 1 \pmod{p}$
- ▶ $x * x^{p-2} \equiv 1 \pmod{p}$
- ▶ $x^{p-2} = \text{invert}(x, p)$

```
static_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = "(((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 0;

loop:
    if (is_single_word) {
        BN_ULONG rwd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         * 3 as being a prime number because it's a multiple of
         * three.
         * 2) That it's not a multiple of a known prime. We don't
         * check that rwd-1 is also coprime to all the known
         * primes because there aren't many small primes where
         * that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rwd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) == 1 (except for 2) */
            if ((BN_ULONG)mods[i] + delta < primes[i]) {
                delta += 2;
            }
        }
    }
}
```

Multiplication mod 13

$$x * x^{p-2} \equiv 1 \pmod{p}$$

*	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	2	4	6	8	10	12	1	3	5	7	9	11
3	0	3	6	9	12	2	5	8	11	1	4	7	10
4	0	4	8	12	3	7	11	2	6	10	1	5	9
5	0	5	10	2	7	12	4	9	1	6	11	3	8
6	0	6	12	5	11	4	10	3	9	2	8	1	7
7	0	7	1	8	2	9	3	10	4	11	5	12	6
8	0	8	3	11	6	1	9	4	12	7	2	10	5
9	0	9	5	1	10	6	2	11	7	3	12	8	4
10	0	10	7	4	1	11	8	5	2	12	9	6	3
11	0	11	9	7	5	3	1	12	10	8	6	4	2
12	0	12	11	10	9	8	7	6	5	4	3	2	1

```

static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

    again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            /* mod 0;
            mods[i] = (uint16_t)mod;
            if (bits is so small that it fits into a single word then we
            additionally don't want to exceed that many bits. */
            if (is_single_word) {
                BN_ULONG size_limit;
                if (bits <= BN_BITS2) {
                    /* avoid undefined behavior. */
                    size_limit = (((BN_ULONG)0) - get_word(rnd);
                } else
                    size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
                if (size_limit < maxdelta) {
                    maxdelta = size_limit;
                }
            }
            delta = 0;
            if (is_single_word) {
                BN_ULONG rnd_word = get_word(rnd);
                /* the case that the candidate prime is a single word then
                we check that:
                1) It's greater than primes[i] because we shouldn't reject
                3 as being a prime number because it's a multiple of
                three.
                2) That it's not a multiple of a known prime. We don't
                check that rnd-1 is also coprime to all the known
                primes because there aren't many small primes where
                that's true. */
                for (j = 1; j < NUMPRIMES && primes[j] < rnd_word; j++) {
                    if ((mods[j] + delta) % primes[j] == 0) {
                        delta += 2;
                        if (delta > maxdelta) {
                            goto again;
                        }
                    }
                }
            } else {
                for (j = 1; j < NUMPRIMES; j++) {
                    /* check that rnd is not a prime and also
                    * that gcd(rnd-1, primes) == 1 (except for 2) */
                    if ((mods[j] + delta) % primes[j] == 0) {
                        delta += 2;
                    }
                }
            }
        }
    }
}

```

Exponentiation mod 13

$$x * x^{p-2} \equiv 1 \pmod{p}$$

x^y	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	2	4	8	3	6	12	11	9	5	10	7	1
3	1	3	9	1	3	9	1	3	9	1	3	9	1
4	1	4	3	12	9	10	1	4	3	12	9	10	1
5	1	5	12	8	1	5	12	8	1	5	12	8	1
6	1	6	10	8	9	2	12	7	3	5	4	11	1
7	1	7	10	5	9	11	12	6	3	8	4	2	1
8	1	8	12	5	1	8	12	5	1	8	12	5	1
9	1	9	3	1	9	3	1	9	3	1	9	3	1
10	1	10	9	12	3	4	1	10	9	12	3	4	1
11	1	11	4	5	3	7	12	2	9	8	10	6	1
12	1	12	1	12	1	12	1	12	1	12	1	12	1

```

static prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == ((BN_ULONG)-1)) {
            /* random 0; */
            continue;
        }
        mods[i] = (uint16_t)mod;

        /* If bits is so small that it fits into a single word then we
         * additionally don't want to exceed that many bits. */
        if (is_single_word) {
            BN_ULONG size_limit;
            if (bits == BN_BITS2) {
                /* Avoid undefined behavior. */
                size_limit = (((BN_ULONG)0) - get_word(rnd));
            } else {
                size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
            }
            if (size_limit < maxdelta) {
                maxdelta = size_limit;
            }
            delta = 0;
        } else {
            /* (is_single_word) {
             * BN_ULONG rnd_word = get_word(rnd);
             *
             * /* In the case that the candidate prime is a single word then
             * /* we check that:
             * /* 1) It's greater than primes[i] because we shouldn't reject
             * /* 3 as being a prime number because it's a multiple of
             * /* three.
             * /* 2) That it's not a multiple of a known prime. We don't
             * /* check that rnd-1 is also coprime to all the known
             * /* primes because there aren't many small primes where
             * /* that's true. */
             * for (j = 1; j < NUMPRIMES && primes[j] < rnd_word; j++) {
             *     if ((mods[j] + delta) % primes[j] == 0) {
             *         delta += 2;
             *         if (delta > maxdelta) {
             *             goto again;
             *         }
             *     }
             * }
             * goto loop;
             * }
             * } else {
             *     for (i = 1; i < NUMPRIMES; i++) {
             *         /* check that rnd is not a prime and also
             *         /* that get(rnd-1,primes) == 1 (except for 2) */
             *         if ((mods[i] + delta) % primes[i] == 0) {
             *             delta += 2;
             *             if (delta > maxdelta) {
             *                 goto again;
             *             }
             *         }
             *     }
             *     goto loop;
             * }
             * }

```

Euler's totient theorem and phi

- ▶ $\gcd(x, n) = 1 \rightarrow x^{\varphi(n)} \equiv 1 \pmod{n}$
- ▶ $\varphi(n) = |\{k | 1 \leq k < n \wedge \gcd(k, n) = 1\}|$
- ▶ def phi(n):
 return len([k for k in range(1, n) if $\gcd(k, n) == 1$])
- ▶ This definition is inefficient to calculate (linear in the value of n , so exponential in the bitlength of n)

```

static inline probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            delta = 0;
        } else {
            delta = size_limit;
        }
    } else {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                }
                goto again;
            }
            goto loop;
        }
    }
} else {
    for (i = 1; i < NUMPRIMES; i++) {
        /* check that rnd is not a prime and also
         * that gcd(rnd-1, primes) == 1 (except for 2) */
        if ((BN_ULONG)rnd_word % primes[i] <= 1) {
            delta += 2;
        }
    }
}

```

Fundamental Theorem of Arithmetic

- Any integer can be decomposed (uniquely) into a product of prime powers
- $\forall n \exists \vec{v} (n = \prod_{i=1}^{\text{len}(\vec{v})} p_i^{v_i})$
- e.g. for $n = 84 = 2 * 42 = 2^2 * 21 = 2^2 * 3 * 7$ $\vec{v} = [2, 1, 0, 1]$
- Computing \vec{v} from n is expensive (algorithms like GNFS and ECM find \vec{v} slower than polytime in the bitlength of n)
- Computing n from \vec{v} is fast (p_i can be found via sieving, and multiplication is cheap)
- When coding, a sparse representation of the prime vector is more convenient:

```

histogram = lambda s: (lambda d: [[d.__setitem__(c, d.get(c, 0)+1) for c in s], d][-1])(dict())
product = lambda xs: reduce(__import__('operator').mul, xs, 1)
assert product([2,2,3,7]) == 84
assert histogram([2,2,3,7]) == {2: 2, 3: 1, 7: 1}

```

```

static inline probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we know a random number 'rnd' to test. */
    BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
    if (mod == (BN_ULONG)-1) {
        return 0;
    }
    mods[i] = (uint16_t)mod;

    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        /* we want to find a prime of size <= (BN_BITS2 - 1) bits. */
        size_limit = (((BN_ULONG)0) - get_word(rnd));
    } else {
        /* we want to find a prime of size <= (bits - 1) bits. */
        size_limit = (BN_ULONG)bits - 1;
    }
    if (size_limit < maxdelta) {
        /* we want to find a prime of size <= (bits - 1) bits. */
        delta = 0;
    } else {
        delta = 0;
    }
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's not a known prime. If it is we shouldn't reject
         *    it as a prime value because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that would be a problem.
         */
        for (i = 1; i < NUMPRIMES; i++) {
            if (mods[i] + delta % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if ((rnd_word + delta % primes[i]) <= 1) {
                delta += 2;
            }
        }
    }
}

```

Calculating Euler's phi via FTA

If we know the factors of n , we can compute $\varphi(n)$ efficiently:

$$1. \varphi(p^k) = (p - 1) * p^{k-1}$$

$$2. \gcd(n, m) = 1 \rightarrow \varphi(n * m) = \varphi(n) * \varphi(m)$$

- ▶ $\varphi(n) = \varphi(\prod_{i=1}^{\text{len}(v)} p_i^{v_i}) = \prod_{i=1}^{\text{len}(v)} \varphi(p_i^{v_i}) = \prod_{i=1}^{\text{len}(v)} (p_i^{v_i} - 1) * p_i^{v_i-1}$
- ▶ e.g. $\varphi(82) = \varphi(2^2 * 3 * 7) = \varphi(2^2) * \varphi(3) * \varphi(7) = (2 - 1) * 2^1 * (3 - 1) * (7 - 1) = 1 * 2 * 2 * 6 = 24$
- ▶

```

histogram = lambda s: (lambda d: [[d.__setitem__(c, d.get(c, 0) + 1) for c in s], d][-1])(dict())
product = lambda xs: reduce(__import__('operator').mul, xs, 1)
def phi(factors_of_n):
    return product([(p-1)*(p**(k-1)) for (p, k) in histogram(factors_of_n).items()])

assert product([2,2,3,7]) == 84 and phi([2,2,3,7]) == 24
    
```

```

static inline PROBABLE_PRIME(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod < primes[i])
            mods[i] = (uint16_t)mod;
        /* If bits is so small that it fits into a single word then we
         * additionally don't want to exceed that many bits. */
        if (is_single_word) {
            BN_ULONG size_limit;
            if (bits == BN_BITS2) {
                /* avoid undefined behavior. */
                size_limit = "((BN_ULONG)0) - get_word(rnd);
            } else
                size_limit = (BN_ULONG)1 < (bits) ? (bits) - 1 :
                    if (size_limit < maxdelta) {
                        maxdelta = size_limit;
                        delta = 0;
                    }
            if (is_single_word) {
                BN_ULONG rnd_word = get_word(rnd);
                /* In the case that the candidate prime is a single word then
                 * 1) It's greater than primes[i] because we shouldn't reject
                 *    3 as being a prime number because it's a multiple of
                 *    2) That it's not a multiple of a known prime. We don't
                 *    check that rnd-1 is also coprime to all the known
                 *    primes because there aren't many small primes where
                 *    that's true. */
                for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
                    if ((mods[i] + delta) % primes[i] == 0) {
                        delta += 2;
                        if (delta > maxdelta) {
                            goto again;
                        }
                        goto loop;
                    }
                }
            } else {
                for (i = 1; i < NUMPRIMES; i++) {
                    /* check that rnd is not a prime and also
                     * that gcd(rnd-1, primes) = 1 (except for 2) */
                    if ((mods[i] + delta) % primes[i] <= 1)
                        delta += 2;
                }
            }
        }
    }
}
    
```


Inverses modulo a composite

- from gmpy import gcd, invert

```
def eee(x, y):
    x, y = min(x, y), max(x, y)
    r, s, t = x, 1, 0
    R, S, T = y, 0, 1
    while r > 0 and R > 0:
        q = r/R
        new = r-q*R, s-q*S, t-q*T
        r, s, t = R, S, T
        R, S, T = new
    assert gcd(x, y) == r # gcd from euclidean algorithm
    assert r == x*s + y*t # s and t are the bezout coefficients
    inv = s + y*(s < 0) # modular inverse from bezout coefficient
    if r == 1:
        assert (x * inv) % y == 1
    return (r, s, t, inv)
```

```
assert [eee(i, 7)[3] for i in range(1, 7)]
assert [invert(i, 7) for i in range(1, 7)]
```

```
static prime_prob(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 0;
loop:
    BN_ULONG rnd_word = get_word(rnd);
    /* In the case that the candidate prime is a single word then
     * we check that:
     * 1) It's greater than primes[i] because we shouldn't reject
     * 3 as being a prime number because it's a multiple of
     * three.
     * 2) That it's not a multiple of a known prime. We don't
     * check that rnd-1 is also coprime to all the known
     * primes because there aren't many small primes where
     * that's true. */
    for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
        if (mods[i] && delta % primes[i] == 0) {
            delta += primes[i];
            goto loop;
        }
    }
    else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if ((BN_ULONG)rnd_word < primes[i] && delta % primes[i] == 0) {
                delta += primes[i];
                goto loop;
            }
        }
    }
}
```

Correctness of RSA¹

- ▶ $e * d \equiv 1 \pmod{\varphi(n)}$
- ▶ $\text{enc}(x) \equiv x^e \pmod{n}$
- ▶ $\text{dec}(x) \equiv x^d \pmod{n}$
- ▶ $\text{dec}(\text{enc}(x)) \equiv (x^e)^d \equiv x^{e*d} \equiv x^1 \pmod{n}$ (Euler)

```
static int probabilistic_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 0;
loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * a that rnd-1 is prime = 1 (except for 2) */
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
    return 1;
}
```

¹ Assumes $\gcd(x, n) = 1$, full proof is a little bit more involved

Exploiting RSA: Cube root of small message

```

▶ from codecs import encode
from gmpy import invert, next_prime
import os

d = 0
while d == 0:
    p = next_prime(int(encode(os.urandom(1024/8), 'hex'), 16))
    q = next_prime(int(encode(os.urandom(1024/8), 'hex'), 16))
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 3
    d = invert(e, phi)

```

```

message = int(encode('hello', 'hex'), 16)
ciphertext = pow(message, e, n)
assert pow(ciphertext, d, n) == message
assert round(ciphertext**(1.0/3)) == message

```

```

static inline BN_ULONG *rnd, int bits) {
    int i;
    while (i < mods[NUMPRIMES]) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (<is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (BN_ULONG)0;
        } else {
            size_limit = ((BN_ULONG)1) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
        delta = 0;
    }
loop:
    if (<is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                }
                goto again;
            }
            goto loop;
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if ((mod + delta) % primes[i] == 0) {
                delta += 2;
            }
        }
    }
}

```

Resources

- ▶ [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))
- ▶ https://en.wikipedia.org/wiki/Extended_Euclidean_Algorithm
- ▶ https://en.wikipedia.org/wiki/Modular_arithmetic
- ▶ <https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf>
- ▶ <https://cryptopals.com/>, Sets 5 and 6

```
static prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == ((BN_ULONG)-1)) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (bits == BN_BITS2) {
        /* Avoid undefined behavior. */
        /* size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
         * if (size_limit < 0) size_limit = 0;
         * maxdelta = size_limit;
         */
    } else {
        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         * 3 as being a prime number because it's a multiple of
         * three.
         * 2) That it's not a multiple of a known prime. We don't
         * check that rnd-1 is also coprime to all the known
         * primes because there aren't many small primes where
         * that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) == 1 (except for 2) */
            if ((mods[i] + delta) % primes[i] <= 1) {
                delta += 2;
            }
        }
    }
}
```