class Bignum

Bignum objects hold integers outside the range of Fixnum. Bignum objects are created automatically when integer calculations would otherwise overflow a Fixnum. When a calculation involving Bignum objects returns a result that will fit in a Fixnum, the result is automatically converted.

For the purposes of the bitwise operations and [], a Bignum is treated as if it were an infinite-length bitstring with 2's complement representation.

While Fixnum values are immediate, Bignum objects are not—assignment and parameter passing work with references to objects, not the objects themselves.

In Files

```
bignum.c
```

Parent

Integer

Constants

```
GMP_VERSION
```

The version of loaded GMP.

Public Instance Methods

big % other → Numeric

modulo(other) → Numeric

Returns big modulo other. See Numeric#divmod for more information.

```
VALUE
rb_big_modulo(VALUE x, VALUE y)
{
    VALUE z;

    if (FIXNUM_P(y)) {
        y = rb_int2big(FIX2LONG(y));
    }
    else if (!RB_BIGNUM_TYPE_P(y)) {
        return rb_num_coerce_bin(x, y, '%');
    }
```

```
    bigdivmod(x, y, 0, &z);

    return bignorm(z);
}
```

big & numeric → integer

Performs bitwise and between big and numeric.

```
VALUE
rb_big_and(VALUE x, VALUE y)
{
    VALUE z;
    BDIGIT *ds1, *ds2, *zds;
    long i, xn, yn, n1, n2;
    BDIGIT hibitsx, hibitsy;
    BDIGIT hibits1, hibits2;
    VALUE tmpv;
    BDIGIT tmph;
    long tmpn;

    if (!FIXNUM_P(y) && !RB_BIGNUM_TYPE_P(y)) {
        return rb_num_coerce_bit(x, y, '&');
    }

    hibitsx = abs2twocomp(&x, &xn);
    if (FIXNUM_P(y)) {
        return bigand_int(x, xn, hibitsx, FIX2LONG(y));
    }
    hibitsy = abs2twocomp(&y, &yn);
    if (xn > yn) {
        tmpv = x; x = y; y = tmpv;
        tmpn = xn; xn = yn; yn = tmpn;
        tmph = hibitsx; hibitsx = hibitsy; hibitsy = tmph;
    }
    n1 = xn;
    n2 = yn;
    ds1 = BDIGITS(x);
    ds2 = BDIGITS(y);
    hibits1 = hibitsx;
    hibits2 = hibitsy;

    if (!hibits1)
        n2 = n1;
```

```
    z = bignew(n2, 0);
    zds = BDIGITS(z);

    for (i=0; i<n1; i++) {
        zds[i] = ds1[i] & ds2[i];
    }
    for (; i<n2; i++) {
        zds[i] = hibits1 & ds2[i];
    }
    twocomp2abs_bang(z, hibits1 && hibits2);
    RB_GC_GUARD(x);
    RB_GC_GUARD(y);
    return bignorm(z);
}
```

big * other → Numeric

Multiplies big and other, returning the result.

```
VALUE
rb_big_mul(VALUE x, VALUE y)
{
    if (FIXNUM_P(y)) {
        y = rb_int2big(FIX2LONG(y));
    }
    else if (RB_BIGNUM_TYPE_P(y)) {
    }
    else if (RB_FLOAT_TYPE_P(y)) {
        return DBL2NUM(rb_big2dbl(x) * RFLOAT_VALUE(y));
    }
    else {
        return rb_num_coerce_bin(x, y, '*');
    }

    return bignorm(bigmul0(x, y));
}
```

big ** exponent → numeric

Raises big to the exponent power (which may be an integer, float, or anything that will coerce to a number). The result may be a Fixnum, Bignum, or Float

```
123456789 ** 2      #=> 15241578750190521
123456789 ** 1.2    #=> 5126464716.09932
123456789 ** -2     #=> 6.5610001194102e-17
```

```
VALUE
rb_big_pow(VALUE x, VALUE y)
{
    double d;
    SIGNED_VALUE yy;

  again:
    if (y == INT2FIX(0)) return INT2FIX(1);
    if (RB_FLOAT_TYPE_P(y)) {
        d = RFLOAT_VALUE(y);
        if ((!BIGNUM_SIGN(x) && !BIGZEROP(x)) && d != round(d))
            return rb_funcall(rb_complex_raw1(x), rb_intern("**"), 1, y);
    }
    else if (RB_BIGNUM_TYPE_P(y)) {
        y = bignorm(y);
        if (FIXNUM_P(y))
            goto again;
        rb_warn("in a**b, b may be too big");
        d = rb_big2dbl(y);
    }
    else if (FIXNUM_P(y)) {
        yy = FIX2LONG(y);

        if (yy < 0)
            return rb_funcall(rb_rational_raw1(x), rb_intern("**"), 1, y);
        else {
            VALUE z = 0;
            SIGNED_VALUE mask;
            const size_t xbits = rb_absint_numwords(x, 1, NULL);
            const size_t BIGLEN_LIMIT = 32*1024*1024;

            if (xbits == (size_t)-1 ||
                (xbits > BIGLEN_LIMIT) ||
                (xbits * yy > BIGLEN_LIMIT)) {
                rb_warn("in a**b, b may be too big");
                d = (double)yy;
            }
            else {
                for (mask = FIXNUM_MAX + 1; mask; mask >>= 1) {
                    if (z) z = bigsq(z);
                    if (yy & mask) {
                        z = z ? bigtrunc(bigmul0(z, x)) : x;
                    }
                }
                return bignorm(z);
```

```
            }
        }
    }
    else {
        return rb_num_coerce_bin(x, y, rb_intern("**"));
    }
    return DBL2NUM(pow(rb_big2dbl(x), d));
}
```

big + other → Numeric

Adds big and other, returning the result.

```
VALUE
rb_big_plus(VALUE x, VALUE y)
{
    long n;

    if (FIXNUM_P(y)) {
        n = FIX2LONG(y);
        if ((n > 0) != BIGNUM_SIGN(x)) {
            if (n < 0) {
                n = -n;
            }
            return bigsub_int(x, n);
        }
        if (n < 0) {
            n = -n;
        }
        return bigadd_int(x, n);
    }
    else if (RB_BIGNUM_TYPE_P(y)) {
        return bignorm(bigadd(x, y, 1));
    }
    else if (RB_FLOAT_TYPE_P(y)) {
        return DBL2NUM(rb_big2dbl(x) + RFLOAT_VALUE(y));
    }
    else {
        return rb_num_coerce_bin(x, y, '+');
    }
}
```

big - other → Numeric

Subtracts other from big, returning the result.

```
VALUE
rb_big_minus(VALUE x, VALUE y)
{
    long n;

    if (FIXNUM_P(y)) {
        n = FIX2LONG(y);
        if ((n > 0) != BIGNUM_SIGN(x)) {
            if (n < 0) {
                n = -n;
            }
            return bigadd_int(x, n);
        }
        if (n < 0) {
            n = -n;
        }
        return bigsub_int(x, n);
    }
    else if (RB_BIGNUM_TYPE_P(y)) {
        return bignorm(bigadd(x, y, 0));
    }
    else if (RB_FLOAT_TYPE_P(y)) {
        return DBL2NUM(rb_big2dbl(x) - RFLOAT_VALUE(y));
    }
    else {
        return rb_num_coerce_bin(x, y, '-');
    }
}
```

-big → integer

Unary minus (returns an integer whose value is 0-big)

```
VALUE
rb_big_uminus(VALUE x)
{
    VALUE z = rb_big_clone(x);

    BIGNUM_SET_SIGN(z, !BIGNUM_SIGN(x));

    return bignorm(z);
}
```

big / other → Numeric

Performs division: the class of the resulting object depends on the class of numeric and on the magnitude of the result.

```
VALUE
rb_big_div(VALUE x, VALUE y)
{
    return rb_big_divide(x, y, '/');
}
```

big < real → true or false

Returns true if the value of big is less than that of real.

```
static VALUE
big_lt(VALUE x, VALUE y)
{
    return big_op(x, y, big_op_lt);
}
```

big << numeric → integer

Shifts big left numeric positions (right if numeric is negative).

```
VALUE
rb_big_lshift(VALUE x, VALUE y)
{
    int lshift_p;
    size_t shift_numdigits;
    int shift_numbits;

    for (;;) {
        if (FIXNUM_P(y)) {
            long l = FIX2LONG(y);
            unsigned long shift;
            if (0 <= l) {
                lshift_p = 1;
                shift = l;
            }
            else {
                lshift_p = 0;
                shift = 1+(unsigned long)(-(l+1));
            }
            shift_numbits = (int)(shift & (BITSPERDIG-1));
            shift_numdigits = shift >> bit_length(BITSPERDIG-1);
            return bignorm(big_shift3(x, lshift_p, shift_numdigits, shift_numbits));
        }
```

```
        else if (RB_BIGNUM_TYPE_P(y)) {
            return bignorm(big_shift2(x, 1, y));
        }
        y = rb_to_int(y);
    }
}
```

big <= real → true or false

Returns true if the value of big is less than or equal to that of real.

```
static VALUE
big_le(VALUE x, VALUE y)
{
    return big_op(x, y, big_op_le);
}
```

big <=> numeric → -1, 0, +1 or nil

Comparison—Returns -1, 0, or +1 depending on whether big is less than, equal to, or greater than numeric. This is the basis for the tests in Comparable.

nil is returned if the two values are incomparable.

```
VALUE
rb_big_cmp(VALUE x, VALUE y)
{
    int cmp;

    if (FIXNUM_P(y)) {
        x = bignorm(x);
        if (FIXNUM_P(x)) {
            if (FIX2LONG(x) > FIX2LONG(y)) return INT2FIX(1);
            if (FIX2LONG(x) < FIX2LONG(y)) return INT2FIX(-1);
            return INT2FIX(0);
        }
        else {
            if (BIGNUM_NEGATIVE_P(x)) return INT2FIX(-1);
            return INT2FIX(1);
        }
    }
    else if (RB_BIGNUM_TYPE_P(y)) {
    }
    else if (RB_FLOAT_TYPE_P(y)) {
        return rb_integer_float_cmp(x, y);
    }
```

```
    else {
        return rb_num_coerce_cmp(x, y, rb_intern("<=>"));
    }

    if (BIGNUM_SIGN(x) > BIGNUM_SIGN(y)) return INT2FIX(1);
    if (BIGNUM_SIGN(x) < BIGNUM_SIGN(y)) return INT2FIX(-1);

    cmp = bary_cmp(BDIGITS(x), BIGNUM_LEN(x), BDIGITS(y), BIGNUM_LEN(y));
    if (BIGNUM_SIGN(x))
        return INT2FIX(cmp);
    else
        return INT2FIX(-cmp);
}
```

big == obj → true or false

Returns true only if obj has the same value as big. Contrast this with Bignum#eql?, which requires obj to be a Bignum.

```
68719476736 == 68719476736.0   #=> true
```

```
VALUE
rb_big_eq(VALUE x, VALUE y)
{
    if (FIXNUM_P(y)) {
        if (bignorm(x) == y) return Qtrue;
        y = rb_int2big(FIX2LONG(y));
    }
    else if (RB_BIGNUM_TYPE_P(y)) {
    }
    else if (RB_FLOAT_TYPE_P(y)) {
        return rb_integer_float_eq(x, y);
    }
    else {
        return rb_equal(y, x);
    }
    if (BIGNUM_SIGN(x) != BIGNUM_SIGN(y)) return Qfalse;
    if (BIGNUM_LEN(x) != BIGNUM_LEN(y)) return Qfalse;
    if (MEMCMP(BDIGITS(x),BDIGITS(y),BDIGIT,BIGNUM_LEN(y)) != 0) return Qfalse;
    return Qtrue;
}
```

big > real → true or false

Returns true if the value of big is greater than that of real.

```
static VALUE
big_gt(VALUE x, VALUE y)
{
    return big_op(x, y, big_op_gt);
}
```

big >= real → true or false

Returns true if the value of big is greater than or equal to that of real.

```
static VALUE
big_ge(VALUE x, VALUE y)
{
    return big_op(x, y, big_op_ge);
}
```

big >> numeric → integer

Shifts big right numeric positions (left if numeric is negative).

```
VALUE
rb_big_rshift(VALUE x, VALUE y)
{
    int lshift_p;
    size_t shift_numdigits;
    int shift_numbits;

    for (;;) {
        if (FIXNUM_P(y)) {
            long l = FIX2LONG(y);
            unsigned long shift;
            if (0 <= l) {
                lshift_p = 0;
                shift = l;
            }
            else {
                lshift_p = 1;
                shift = 1+(unsigned long)(-(l+1));
            }
            shift_numbits = (int)(shift & (BITSPERDIG-1));
            shift_numdigits = shift >> bit_length(BITSPERDIG-1);
            return bignorm(big_shift3(x, lshift_p, shift_numdigits, shift_numbits));
        }
        else if (RB_BIGNUM_TYPE_P(y)) {
            return bignorm(big_shift2(x, 0, y));
```

```
        }
        y = rb_to_int(y);
    }
}
```

big[n] → 0, 1

Bit Reference—Returns the nth bit in the (assumed) binary representation of big, where big is the least significant bit.

```
a = 9**15
50.downto(0) do |n|
  print a[n]
end
```

produces:

```
000101110110100000111000011110010100111100010111001
```

```
static VALUE
rb_big_aref(VALUE x, VALUE y)
{
    BDIGIT *xds;
    size_t shift;
    size_t i, s1, s2;
    long l;
    BDIGIT bit;

    if (RB_BIGNUM_TYPE_P(y)) {
        if (!BIGNUM_SIGN(y))
            return INT2FIX(0);
        bigtrunc(y);
        if (BIGSIZE(y) > sizeof(size_t)) {
          out_of_range:
            return BIGNUM_SIGN(x) ? INT2FIX(0) : INT2FIX(1);
        }
#if SIZEOF_SIZE_T <= SIZEOF_LONG
        shift = big2ulong(y, "long");
#else
        shift = big2ull(y, "long long");
#endif
    }
    else {
        l = NUM2LONG(y);
        if (l < 0) return INT2FIX(0);
```

```
        shift = (size_t)l;
    }
    s1 = shift/BITSPERDIG;
    s2 = shift%BITSPERDIG;
    bit = (BDIGIT)1 << s2;

    if (s1 >= BIGNUM_LEN(x)) goto out_of_range;

    xds = BDIGITS(x);
    if (BIGNUM_POSITIVE_P(x))
        return (xds[s1] & bit) ? INT2FIX(1) : INT2FIX(0);
    if (xds[s1] & (bit-1))
        return (xds[s1] & bit) ? INT2FIX(0) : INT2FIX(1);
    for (i = 0; i < s1; i++)
        if (xds[i])
            return (xds[s1] & bit) ? INT2FIX(0) : INT2FIX(1);
    return (xds[s1] & bit) ? INT2FIX(1) : INT2FIX(0);
}
```

big ^ numeric → integer

Performs bitwise +exclusive or+ between big and numeric.

```
VALUE
rb_big_xor(VALUE x, VALUE y)
{
    VALUE z;
    BDIGIT *ds1, *ds2, *zds;
    long i, xn, yn, n1, n2;
    BDIGIT hibitsx, hibitsy;
    BDIGIT hibits1, hibits2;
    VALUE tmpv;
    BDIGIT tmph;
    long tmpn;

    if (!FIXNUM_P(y) && !RB_BIGNUM_TYPE_P(y)) {
        return rb_num_coerce_bit(x, y, '^');
    }

    hibitsx = abs2twocomp(&x, &xn);
    if (FIXNUM_P(y)) {
        return bigxor_int(x, xn, hibitsx, FIX2LONG(y));
    }
    hibitsy = abs2twocomp(&y, &yn);
    if (xn > yn) {
```

```
        tmpv = x; x = y; y = tmpv;
        tmpn = xn; xn = yn; yn = tmpn;
        tmph = hibitsx; hibitsx = hibitsy; hibitsy = tmph;
    }
    n1 = xn;
    n2 = yn;
    ds1 = BDIGITS(x);
    ds2 = BDIGITS(y);
    hibits1 = hibitsx;
    hibits2 = hibitsy;

    z = bignew(n2, 0);
    zds = BDIGITS(z);

    for (i=0; i<n1; i++) {
        zds[i] = ds1[i] ^ ds2[i];
    }
    for (; i<n2; i++) {
        zds[i] = hibitsx ^ ds2[i];
    }
    twocomp2abs_bang(z, (hibits1 ^ hibits2) != 0);
    RB_GC_GUARD(x);
    RB_GC_GUARD(y);
    return bignorm(z);
}
```

abs → aBignum

magnitude → aBignum

Returns the absolute value of big.

```
-1234567890987654321.abs   #=> 1234567890987654321
```

```
static VALUE
rb_big_abs(VALUE x)
{
    if (!BIGNUM_SIGN(x)) {
        x = rb_big_clone(x);
        BIGNUM_SET_SIGN(x, 1);
    }
    return x;
}
```

bit_length → integer

Returns the number of bits of the value of int.

“the number of bits” means that the bit position of the highest bit which is different to the sign bit. (The bit position of the bit 2**n is n+1.) If there is no such bit (zero or minus one), zero is returned.

I.e. This method returns ceil(log2(int < 0 ? -int : int+1)).

```
(-2**10000-1).bit_length  #=> 10001
(-2**10000).bit_length    #=> 10000
(-2**10000+1).bit_length  #=> 10000

(-2**1000-1).bit_length   #=> 1001
(-2**1000).bit_length     #=> 1000
(-2**1000+1).bit_length   #=> 1000

(2**1000-1).bit_length    #=> 1000
(2**1000).bit_length      #=> 1001
(2**1000+1).bit_length    #=> 1001

(2**10000-1).bit_length   #=> 10000
(2**10000).bit_length     #=> 10001
(2**10000+1).bit_length   #=> 10001
```

This method can be used to detect overflow in Array#pack as follows.

```
if n.bit_length < 32
  [n].pack("l") # no overflow
else
  raise "overflow"
end
```

```
static VALUE
rb_big_bit_length(VALUE big)
{
    int nlz_bits;
    size_t numbytes;

    static const BDIGIT char_bit[1] = { CHAR_BIT };
    BDIGIT numbytes_bary[bdigit_roomof(sizeof(size_t))];
    BDIGIT nlz_bary[1];
    BDIGIT result_bary[bdigit_roomof(sizeof(size_t)+1)];

    numbytes = rb_absint_size(big, &nlz_bits);

    if (numbytes == 0)
```

```
        return LONG2FIX(0);

    if (BIGNUM_NEGATIVE_P(big) && rb_absint_singlebit_p(big)) {
        if (nlz_bits != CHAR_BIT-1) {
            nlz_bits++;
        }
        else {
            nlz_bits = 0;
            numbytes--;
        }
    }

    if (numbytes <= SIZE_MAX / CHAR_BIT) {
        return SIZET2NUM(numbytes * CHAR_BIT - nlz_bits);
    }

    nlz_bary[0] = nlz_bits;

    bary_unpack(BARY_ARGS(numbytes_bary), &numbytes, 1, sizeof(numbytes), 0,
            INTEGER_PACK_NATIVE_BYTE_ORDER);
    BARY_SHORT_MUL(result_bary, numbytes_bary, char_bit);
    BARY_SUB(result_bary, result_bary, nlz_bary);

    return rb_integer_unpack(result_bary, numberof(result_bary), sizeof(BDIGIT), 0,
            INTEGER_PACK_LSWORD_FIRST|INTEGER_PACK_NATIVE_BYTE_ORDER);
}
```

coerce(numeric) → array

Returns an array with both a numeric and a big represented as Bignum objects.

This is achieved by converting numeric to a Bignum.

A TypeError is raised if the numeric is not a Fixnum or Bignum type.

```
(0x3FFFFFFFFFFFFFFF+1).coerce(42)   #=> [42, 4611686018427387904]
```

```
static VALUE
rb_big_coerce(VALUE x, VALUE y)
{
    if (FIXNUM_P(y)) {
        y = rb_int2big(FIX2LONG(y));
    }
    else if (!RB_BIGNUM_TYPE_P(y)) {
        rb_raise(rb_eTypeError, "can't coerce %"PRIsVALUE" to Bignum",
                 rb_obj_class(y));
```

```
    }
    return rb_assoc_new(y, x);
}
```

div(other) → integer

Performs integer division: returns integer value.

```
VALUE
rb_big_idiv(VALUE x, VALUE y)
{
    return rb_big_divide(x, y, rb_intern("div"));
}
```

divmod(numeric) → array

See Numeric#divmod.

```
VALUE
rb_big_divmod(VALUE x, VALUE y)
{
    VALUE div, mod;

    if (FIXNUM_P(y)) {
        y = rb_int2big(FIX2LONG(y));
    }
    else if (!RB_BIGNUM_TYPE_P(y)) {
        return rb_num_coerce_bin(x, y, rb_intern("divmod"));
    }
    bigdivmod(x, y, &div, &mod);

    return rb_assoc_new(bignorm(div), bignorm(mod));
}
```

eql?(obj) → true or false

Returns true only if obj is a Bignum with the same value as big. Contrast this with Bignum#==, which performs type conversions.

```
68719476736.eql?(68719476736.0)   #=> false
```

```
VALUE
rb_big_eql(VALUE x, VALUE y)
{
    if (!RB_BIGNUM_TYPE_P(y)) return Qfalse;
    if (BIGNUM_SIGN(x) != BIGNUM_SIGN(y)) return Qfalse;
```

```
    if (BIGNUM_LEN(x) != BIGNUM_LEN(y)) return Qfalse;
    if (MEMCMP(BDIGITS(x),BDIGITS(y),BDIGIT,BIGNUM_LEN(y)) != 0) return Qfalse;
    return Qtrue;
}
```

even? → true or false

Returns true if big is an even number.

```
static VALUE
rb_big_even_p(VALUE num)
{
    if (BIGNUM_LEN(num) != 0 && BDIGITS(num)[0] & 1) {
        return Qfalse;
    }
    return Qtrue;
}
```

fdiv(numeric) → float

Returns the floating point result of dividing big by numeric.

```
-1234567890987654321.fdiv(13731)      #=> -89910996357705.5
-1234567890987654321.fdiv(13731.24)   #=> -89909424858035.7
```

```
VALUE
rb_big_fdiv(VALUE x, VALUE y)
{
    double dx, dy;

    dx = big2dbl(x);
    if (FIXNUM_P(y)) {
        dy = (double)FIX2LONG(y);
        if (isinf(dx))
            return big_fdiv_int(x, rb_int2big(FIX2LONG(y)));
    }
    else if (RB_BIGNUM_TYPE_P(y)) {
        dy = rb_big2dbl(y);
        if (isinf(dx) || isinf(dy))
            return big_fdiv_int(x, y);
    }
    else if (RB_FLOAT_TYPE_P(y)) {
        dy = RFLOAT_VALUE(y);
        if (isnan(dy))
            return y;
```

```
        if (isinf(dx))
            return big_fdiv_float(x, y);
    }
    else {
        return rb_num_coerce_bin(x, y, rb_intern("fdiv"));
    }
    return DBL2NUM(dx / dy);
}
```

hash → fixnum

Compute a hash based on the value of big.

See also Object#hash.

```
static VALUE
rb_big_hash(VALUE x)
{
    st_index_t hash;

    hash = rb_memhash(BDIGITS(x), sizeof(BDIGIT)*BIGNUM_LEN(x)) ^ BIGNUM_SIGN(x);
    return INT2FIX(hash);
}
```

inspect(p1 = v1)

Alias for: to_s

abs → aBignum

magnitude → aBignum

Returns the absolute value of big.

```
-1234567890987654321.abs   #=> 1234567890987654321
```

```
static VALUE
rb_big_abs(VALUE x)
{
    if (!BIGNUM_SIGN(x)) {
        x = rb_big_clone(x);
        BIGNUM_SET_SIGN(x, 1);
    }
    return x;
}
```

big % other → Numeric

modulo(other) → Numeric

Returns big modulo other. See Numeric#divmod for more information.

```
VALUE
rb_big_modulo(VALUE x, VALUE y)
{
    VALUE z;

    if (FIXNUM_P(y)) {
        y = rb_int2big(FIX2LONG(y));
    }
    else if (!RB_BIGNUM_TYPE_P(y)) {
        return rb_num_coerce_bin(x, y, '%');
    }
    bigdivmod(x, y, 0, &z);

    return bignorm(z);
}
```

odd? → true or false

Returns true if big is an odd number.

```
static VALUE
rb_big_odd_p(VALUE num)
{
    if (BIGNUM_LEN(num) != 0 && BDIGITS(num)[0] & 1) {
        return Qtrue;
    }
    return Qfalse;
}
```

remainder(numeric) → number

Returns the remainder after dividing big by numeric.

```
-1234567890987654321.remainder(13731)      #=> -6966
-1234567890987654321.remainder(13731.24)   #=> -9906.22531493148
```

```
static VALUE
rb_big_remainder(VALUE x, VALUE y)
{
    VALUE z;

    if (FIXNUM_P(y)) {
```

```
        y = rb_int2big(FIX2LONG(y));
    }
    else if (!RB_BIGNUM_TYPE_P(y)) {
        return rb_num_coerce_bin(x, y, rb_intern("remainder"));
    }
    bigdivrem(x, y, 0, &z);

    return bignorm(z);
}
```

size → integer

Returns the number of bytes in the machine representation of big.

```
(256**10 - 1).size   #=> 12
(256**20 - 1).size   #=> 20
(256**40 - 1).size   #=> 40
```

```
static VALUE
rb_big_size(VALUE big)
{
    return SIZET2NUM(BIGSIZE(big));
}
```

to_f → float

Converts big to a Float. If big doesn't fit in a Float, the result is infinity.

```
static VALUE
rb_big_to_f(VALUE x)
{
    return DBL2NUM(rb_big2dbl(x));
}
```

to_s(base=10) → string

Returns a string containing the representation of big radix base (2 through 36).

```
12345654321.to_s         #=> "12345654321"
12345654321.to_s(2)      #=> "1011011111110110111011110000110001"
12345654321.to_s(8)      #=> "133766736061"
12345654321.to_s(16)     #=> "2dfdbbc31"
78546939656932.to_s(36)  #=> "rubyrules"
```

```
static VALUE
```

```
rb_big_to_s(int argc, VALUE *argv, VALUE x)
{
    int base;

    if (argc == 0) base = 10;
    else {
        VALUE b;

        rb_scan_args(argc, argv, "01", &b);
        base = NUM2INT(b);
    }
    return rb_big2str(x, base);
}
```

Also aliased as: inspect

big | numeric → integer

Performs bitwise or between big and numeric.

```
VALUE
rb_big_or(VALUE x, VALUE y)
{
    VALUE z;
    BDIGIT *ds1, *ds2, *zds;
    long i, xn, yn, n1, n2;
    BDIGIT hibitsx, hibitsy;
    BDIGIT hibits1, hibits2;
    VALUE tmpv;
    BDIGIT tmph;
    long tmpn;

    if (!FIXNUM_P(y) && !RB_BIGNUM_TYPE_P(y)) {
        return rb_num_coerce_bit(x, y, '|');
    }

    hibitsx = abs2twocomp(&x, &xn);
    if (FIXNUM_P(y)) {
        return bigor_int(x, xn, hibitsx, FIX2LONG(y));
    }
    hibitsy = abs2twocomp(&y, &yn);
    if (xn > yn) {
        tmpv = x; x = y; y = tmpv;
        tmpn = xn; xn = yn; yn = tmpn;
        tmph = hibitsx; hibitsx = hibitsy; hibitsy = tmph;
    }
```

```
    n1 = xn;
    n2 = yn;
    ds1 = BDIGITS(x);
    ds2 = BDIGITS(y);
    hibits1 = hibitsx;
    hibits2 = hibitsy;

    if (hibits1)
        n2 = n1;

    z = bignew(n2, 0);
    zds = BDIGITS(z);

    for (i=0; i<n1; i++) {
        zds[i] = ds1[i] | ds2[i];
    }
    for (; i<n2; i++) {
        zds[i] = hibits1 | ds2[i];
    }
    twocomp2abs_bang(z, hibits1 || hibits2);
    RB_GC_GUARD(x);
    RB_GC_GUARD(y);
    return bignorm(z);
}
```

~big → integer

Inverts the bits in big. As Bignums are conceptually infinite length, the result acts as if it had an infinite number of one bits to the left. In hex representations, this is displayed as two periods to the left of the digits.

```
sprintf("%X", ~0x1122334455)    #=> "..FEEDDCCBBAA"
```

```
static VALUE
rb_big_neg(VALUE x)
{
    VALUE z = rb_big_clone(x);
    BDIGIT *ds = BDIGITS(z);
    long n = BIGNUM_LEN(z);

    if (!n) return INT2FIX(-1);

    if (BIGNUM_POSITIVE_P(z)) {
        if (bary_add_one(ds, n)) {
            big_extend_carry(z);
        }
```

```
        BIGNUM_SET_NEGATIVE_SIGN(z);
    }
    else {
        bary_neg(ds, n);
        if (bary_add_one(ds, n))
            return INT2FIX(-1);
        bary_neg(ds, n);
        BIGNUM_SET_POSITIVE_SIGN(z);
    }

    return bignorm(z);
}
```