/\*

\* Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.

\* ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*/

/\*

\* Portions Copyright (c) 1995 Colin Plumb. All rights reserved.

\*/

**package** java.math;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.io.ObjectStreamField;

**import** java.util.Arrays;

**import** java.util.Random;

**import** java.util.concurrent.ThreadLocalRandom;

**import** sun.misc.DoubleConsts;

**import** sun.misc.FloatConsts;

/\*\*

\* Immutable arbitrary-precision integers. All operations behave as if

\* BigIntegers were represented in two's-complement notation (like Java's

\* primitive integer types). BigInteger provides analogues to all of Java's

\* primitive integer operators, and all relevant methods from java.lang.Math.

\* Additionally, BigInteger provides operations for modular arithmetic, GCD

\* calculation, primality testing, prime generation, bit manipulation,

\* and a few other miscellaneous operations.

\*

\* <p>Semantics of arithmetic operations exactly mimic those of Java's integer

\* arithmetic operators, as defined in <i>The Java Language Specification</i>.

\* For example, division by zero throws an {@code ArithmeticException}, and

\* division of a negative by a positive yields a negative (or zero) remainder.

\* All of the details in the Spec concerning overflow are ignored, as

\* BigIntegers are made as large as necessary to accommodate the results of an

\* operation.

\*

\* <p>Semantics of shift operations extend those of Java's shift operators

\* to allow for negative shift distances. A right-shift with a negative

\* shift distance results in a left shift, and vice-versa. The unsigned

\* right shift operator ({@code >>>}) is omitted, as this operation makes

\* little sense in combination with the "infinite word size" abstraction

\* provided by this class.

\*

\* <p>Semantics of bitwise logical operations exactly mimic those of Java's

\* bitwise integer operators. The binary operators ({@code and},

\* {@code or}, {@code xor}) implicitly perform sign extension on the shorter

\* of the two operands prior to performing the operation.

\*

\* <p>Comparison operations perform signed integer comparisons, analogous to

\* those performed by Java's relational and equality operators.

\*

\* <p>Modular arithmetic operations are provided to compute residues, perform

\* exponentiation, and compute multiplicative inverses. These methods always

\* return a non-negative result, between {@code 0} and {@code (modulus - 1)},

\* inclusive.

\*

\* <p>Bit operations operate on a single bit of the two's-complement

\* representation of their operand. If necessary, the operand is sign-

\* extended so that it contains the designated bit. None of the single-bit

\* operations can produce a BigInteger with a different sign from the

\* BigInteger being operated on, as they affect only a single bit, and the

\* "infinite word size" abstraction provided by this class ensures that there

\* are infinitely many "virtual sign bits" preceding each BigInteger.

\*

\* <p>For the sake of brevity and clarity, pseudo-code is used throughout the

\* descriptions of BigInteger methods. The pseudo-code expression

\* {@code (i + j)} is shorthand for "a BigInteger whose value is

\* that of the BigInteger {@code i} plus that of the BigInteger {@code j}."

\* The pseudo-code expression {@code (i == j)} is shorthand for

\* "{@code true} if and only if the BigInteger {@code i} represents the same

\* value as the BigInteger {@code j}." Other pseudo-code expressions are

\* interpreted similarly.

\*

\* <p>All methods and constructors in this class throw

\* {@code NullPointerException} when passed

\* a null object reference for any input parameter.

\*

\* BigInteger must support values in the range

\* -2<sup>{@code Integer.MAX\_VALUE}</sup> (exclusive) to

\* +2<sup>{@code Integer.MAX\_VALUE}</sup> (exclusive)

\* and may support values outside of that range.

\*

\* The range of probable prime values is limited and may be less than

\* the full supported positive range of {@code BigInteger}.

\* The range must be at least 1 to 2<sup>500000000</sup>.

\*

\* **@implNote**

\* BigInteger constructors and operations throw {@code ArithmeticException} when

\* the result is out of the supported range of

\* -2<sup>{@code Integer.MAX\_VALUE}</sup> (exclusive) to

\* +2<sup>{@code Integer.MAX\_VALUE}</sup> (exclusive).

\*

\* **@see** BigDecimal

\* **@author** Josh Bloch

\* **@author** Michael McCloskey

\* **@author** Alan Eliasen

\* **@author** Timothy Buktu

\* **@since** JDK1.1

\*/

**public** **class** BigInteger **extends** Number **implements** Comparable<BigInteger> {

/\*\*

\* The signum of this BigInteger: -1 for negative, 0 for zero, or

\* 1 for positive. Note that the BigInteger zero <i>must</i> have

\* a signum of 0. This is necessary to ensures that there is exactly one

\* representation for each BigInteger value.

\*

\* **@serial**

\*/

**final** **int** signum;

/\*\*

\* The magnitude of this BigInteger, in <i>big-endian</i> order: the

\* zeroth element of this array is the most-significant int of the

\* magnitude. The magnitude must be "minimal" in that the most-significant

\* int ({@code mag[0]}) must be non-zero. This is necessary to

\* ensure that there is exactly one representation for each BigInteger

\* value. Note that this implies that the BigInteger zero has a

\* zero-length mag array.

\*/

**final** **int**[] mag;

// These "redundant fields" are initialized with recognizable nonsense

// values, and cached the first time they are needed (or never, if they

// aren't needed).

/\*\*

\* One plus the bitCount of this BigInteger. Zeros means unitialized.

\*

\* **@serial**

\* **@see** #bitCount

\* **@deprecated** Deprecated since logical value is offset from stored

\* value and correction factor is applied in accessor method.

\*/

@Deprecated

**private** **int** ~~bitCount~~;

/\*\*

\* One plus the bitLength of this BigInteger. Zeros means unitialized.

\* (either value is acceptable).

\*

\* **@serial**

\* **@see** #bitLength()

\* **@deprecated** Deprecated since logical value is offset from stored

\* value and correction factor is applied in accessor method.

\*/

@Deprecated

**private** **int** ~~bitLength~~;

/\*\*

\* Two plus the lowest set bit of this BigInteger, as returned by

\* getLowestSetBit().

\*

\* **@serial**

\* **@see** #getLowestSetBit

\* **@deprecated** Deprecated since logical value is offset from stored

\* value and correction factor is applied in accessor method.

\*/

@Deprecated

**private** **int** ~~lowestSetBit~~;

/\*\*

\* Two plus the index of the lowest-order int in the magnitude of this

\* BigInteger that contains a nonzero int, or -2 (either value is acceptable).

\* The least significant int has int-number 0, the next int in order of

\* increasing significance has int-number 1, and so forth.

\* **@deprecated** Deprecated since logical value is offset from stored

\* value and correction factor is applied in accessor method.

\*/

@Deprecated

**private** **int** ~~firstNonzeroIntNum~~;

/\*\*

\* This mask is used to obtain the value of an int as if it were unsigned.

\*/

**final** **static** **long** ***LONG\_MASK*** = 0xffffffffL;

/\*\*

\* This constant limits {@code mag.length} of BigIntegers to the supported

\* range.

\*/

**private** **static** **final** **int** ***MAX\_MAG\_LENGTH*** = Integer.***MAX\_VALUE*** / Integer.***SIZE*** + 1; // (1 << 26)

/\*\*

\* Bit lengths larger than this constant can cause overflow in searchLen

\* calculation and in BitSieve.singleSearch method.

\*/

**private** **static** **final** **int** ***PRIME\_SEARCH\_BIT\_LENGTH\_LIMIT*** = 500000000;

/\*\*

\* The threshold value for using Karatsuba multiplication. If the number

\* of ints in both mag arrays are greater than this number, then

\* Karatsuba multiplication will be used. This value is found

\* experimentally to work well.

\*/

**private** **static** **final** **int** ***KARATSUBA\_THRESHOLD*** = 80;

/\*\*

\* The threshold value for using 3-way Toom-Cook multiplication.

\* If the number of ints in each mag array is greater than the

\* Karatsuba threshold, and the number of ints in at least one of

\* the mag arrays is greater than this threshold, then Toom-Cook

\* multiplication will be used.

\*/

**private** **static** **final** **int** ***TOOM\_COOK\_THRESHOLD*** = 240;

/\*\*

\* The threshold value for using Karatsuba squaring. If the number

\* of ints in the number are larger than this value,

\* Karatsuba squaring will be used. This value is found

\* experimentally to work well.

\*/

**private** **static** **final** **int** ***KARATSUBA\_SQUARE\_THRESHOLD*** = 128;

/\*\*

\* The threshold value for using Toom-Cook squaring. If the number

\* of ints in the number are larger than this value,

\* Toom-Cook squaring will be used. This value is found

\* experimentally to work well.

\*/

**private** **static** **final** **int** ***TOOM\_COOK\_SQUARE\_THRESHOLD*** = 216;

/\*\*

\* The threshold value for using Burnikel-Ziegler division. If the number

\* of ints in the divisor are larger than this value, Burnikel-Ziegler

\* division may be used. This value is found experimentally to work well.

\*/

**static** **final** **int** ***BURNIKEL\_ZIEGLER\_THRESHOLD*** = 80;

/\*\*

\* The offset value for using Burnikel-Ziegler division. If the number

\* of ints in the divisor exceeds the Burnikel-Ziegler threshold, and the

\* number of ints in the dividend is greater than the number of ints in the

\* divisor plus this value, Burnikel-Ziegler division will be used. This

\* value is found experimentally to work well.

\*/

**static** **final** **int** ***BURNIKEL\_ZIEGLER\_OFFSET*** = 40;

/\*\*

\* The threshold value for using Schoenhage recursive base conversion. If

\* the number of ints in the number are larger than this value,

\* the Schoenhage algorithm will be used. In practice, it appears that the

\* Schoenhage routine is faster for any threshold down to 2, and is

\* relatively flat for thresholds between 2-25, so this choice may be

\* varied within this range for very small effect.

\*/

**private** **static** **final** **int** ***SCHOENHAGE\_BASE\_CONVERSION\_THRESHOLD*** = 20;

/\*\*

\* The threshold value for using squaring code to perform multiplication

\* of a {@code BigInteger} instance by itself. If the number of ints in

\* the number are larger than this value, {@code multiply(this)} will

\* return {@code square()}.

\*/

**private** **static** **final** **int** ***MULTIPLY\_SQUARE\_THRESHOLD*** = 20;

/\*\*

\* The threshold for using an intrinsic version of

\* implMontgomeryXXX to perform Montgomery multiplication. If the

\* number of ints in the number is more than this value we do not

\* use the intrinsic.

\*/

**private** **static** **final** **int** ***MONTGOMERY\_INTRINSIC\_THRESHOLD*** = 512;

// Constructors

/\*\*

\* Translates a byte array containing the two's-complement binary

\* representation of a BigInteger into a BigInteger. The input array is

\* assumed to be in <i>big-endian</i> byte-order: the most significant

\* byte is in the zeroth element.

\*

\* **@param** val big-endian two's-complement binary representation of

\* BigInteger.

\* **@throws** NumberFormatException {@code val} is zero bytes long.

\*/

**public** BigInteger(**byte**[] val) {

**if** (val.length == 0)

**throw** **new** NumberFormatException("Zero length BigInteger");

**if** (val[0] < 0) {

mag = *makePositive*(val);

signum = -1;

} **else** {

mag = *stripLeadingZeroBytes*(val);

signum = (mag.length == 0 ? 0 : 1);

}

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

checkRange();

}

}

/\*\*

\* This private constructor translates an int array containing the

\* two's-complement binary representation of a BigInteger into a

\* BigInteger. The input array is assumed to be in <i>big-endian</i>

\* int-order: the most significant int is in the zeroth element.

\*/

**private** BigInteger(**int**[] val) {

**if** (val.length == 0)

**throw** **new** NumberFormatException("Zero length BigInteger");

**if** (val[0] < 0) {

mag = *makePositive*(val);

signum = -1;

} **else** {

mag = *trustedStripLeadingZeroInts*(val);

signum = (mag.length == 0 ? 0 : 1);

}

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

checkRange();

}

}

/\*\*

\* Translates the sign-magnitude representation of a BigInteger into a

\* BigInteger. The sign is represented as an integer signum value: -1 for

\* negative, 0 for zero, or 1 for positive. The magnitude is a byte array

\* in <i>big-endian</i> byte-order: the most significant byte is in the

\* zeroth element. A zero-length magnitude array is permissible, and will

\* result in a BigInteger value of 0, whether signum is -1, 0 or 1.

\*

\* **@param** signum signum of the number (-1 for negative, 0 for zero, 1

\* for positive).

\* **@param** magnitude big-endian binary representation of the magnitude of

\* the number.

\* **@throws** NumberFormatException {@code signum} is not one of the three

\* legal values (-1, 0, and 1), or {@code signum} is 0 and

\* {@code magnitude} contains one or more non-zero bytes.

\*/

**public** BigInteger(**int** signum, **byte**[] magnitude) {

**this**.mag = *stripLeadingZeroBytes*(magnitude);

**if** (signum < -1 || signum > 1)

**throw**(**new** NumberFormatException("Invalid signum value"));

**if** (**this**.mag.length == 0) {

**this**.signum = 0;

} **else** {

**if** (signum == 0)

**throw**(**new** NumberFormatException("signum-magnitude mismatch"));

**this**.signum = signum;

}

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

checkRange();

}

}

/\*\*

\* A constructor for internal use that translates the sign-magnitude

\* representation of a BigInteger into a BigInteger. It checks the

\* arguments and copies the magnitude so this constructor would be

\* safe for external use.

\*/

**private** BigInteger(**int** signum, **int**[] magnitude) {

**this**.mag = *stripLeadingZeroInts*(magnitude);

**if** (signum < -1 || signum > 1)

**throw**(**new** NumberFormatException("Invalid signum value"));

**if** (**this**.mag.length == 0) {

**this**.signum = 0;

} **else** {

**if** (signum == 0)

**throw**(**new** NumberFormatException("signum-magnitude mismatch"));

**this**.signum = signum;

}

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

checkRange();

}

}

/\*\*

\* Translates the String representation of a BigInteger in the

\* specified radix into a BigInteger. The String representation

\* consists of an optional minus or plus sign followed by a

\* sequence of one or more digits in the specified radix. The

\* character-to-digit mapping is provided by {@code

\* Character.digit}. The String may not contain any extraneous

\* characters (whitespace, for example).

\*

\* **@param** val String representation of BigInteger.

\* **@param** radix radix to be used in interpreting {@code val}.

\* **@throws** NumberFormatException {@code val} is not a valid representation

\* of a BigInteger in the specified radix, or {@code radix} is

\* outside the range from {@link Character#MIN\_RADIX} to

\* {@link Character#MAX\_RADIX}, inclusive.

\* **@see** Character#digit

\*/

**public** BigInteger(String val, **int** radix) {

**int** cursor = 0, numDigits;

**final** **int** len = val.length();

**if** (radix < Character.***MIN\_RADIX*** || radix > Character.***MAX\_RADIX***)

**throw** **new** NumberFormatException("Radix out of range");

**if** (len == 0)

**throw** **new** NumberFormatException("Zero length BigInteger");

// Check for at most one leading sign

**int** sign = 1;

**int** index1 = val.lastIndexOf('-');

**int** index2 = val.lastIndexOf('+');

**if** (index1 >= 0) {

**if** (index1 != 0 || index2 >= 0) {

**throw** **new** NumberFormatException("Illegal embedded sign character");

}

sign = -1;

cursor = 1;

} **else** **if** (index2 >= 0) {

**if** (index2 != 0) {

**throw** **new** NumberFormatException("Illegal embedded sign character");

}

cursor = 1;

}

**if** (cursor == len)

**throw** **new** NumberFormatException("Zero length BigInteger");

// Skip leading zeros and compute number of digits in magnitude

**while** (cursor < len &&

Character.*digit*(val.charAt(cursor), radix) == 0) {

cursor++;

}

**if** (cursor == len) {

signum = 0;

mag = ***ZERO***.mag;

**return**;

}

numDigits = len - cursor;

signum = sign;

// Pre-allocate array of expected size. May be too large but can

// never be too small. Typically exact.

**long** numBits = ((numDigits \* *bitsPerDigit*[radix]) >>> 10) + 1;

**if** (numBits + 31 >= (1L << 32)) {

*reportOverflow*();

}

**int** numWords = (**int**) (numBits + 31) >>> 5;

**int**[] magnitude = **new** **int**[numWords];

// Process first (potentially short) digit group

**int** firstGroupLen = numDigits % *digitsPerInt*[radix];

**if** (firstGroupLen == 0)

firstGroupLen = *digitsPerInt*[radix];

String group = val.substring(cursor, cursor += firstGroupLen);

magnitude[numWords - 1] = Integer.*parseInt*(group, radix);

**if** (magnitude[numWords - 1] < 0)

**throw** **new** NumberFormatException("Illegal digit");

// Process remaining digit groups

**int** superRadix = *intRadix*[radix];

**int** groupVal = 0;

**while** (cursor < len) {

group = val.substring(cursor, cursor += *digitsPerInt*[radix]);

groupVal = Integer.*parseInt*(group, radix);

**if** (groupVal < 0)

**throw** **new** NumberFormatException("Illegal digit");

*destructiveMulAdd*(magnitude, superRadix, groupVal);

}

// Required for cases where the array was overallocated.

mag = *trustedStripLeadingZeroInts*(magnitude);

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

checkRange();

}

}

/\*

\* Constructs a new BigInteger using a char array with radix=10.

\* Sign is precalculated outside and not allowed in the val.

\*/

BigInteger(**char**[] val, **int** sign, **int** len) {

**int** cursor = 0, numDigits;

// Skip leading zeros and compute number of digits in magnitude

**while** (cursor < len && Character.*digit*(val[cursor], 10) == 0) {

cursor++;

}

**if** (cursor == len) {

signum = 0;

mag = ***ZERO***.mag;

**return**;

}

numDigits = len - cursor;

signum = sign;

// Pre-allocate array of expected size

**int** numWords;

**if** (len < 10) {

numWords = 1;

} **else** {

**long** numBits = ((numDigits \* *bitsPerDigit*[10]) >>> 10) + 1;

**if** (numBits + 31 >= (1L << 32)) {

*reportOverflow*();

}

numWords = (**int**) (numBits + 31) >>> 5;

}

**int**[] magnitude = **new** **int**[numWords];

// Process first (potentially short) digit group

**int** firstGroupLen = numDigits % *digitsPerInt*[10];

**if** (firstGroupLen == 0)

firstGroupLen = *digitsPerInt*[10];

magnitude[numWords - 1] = parseInt(val, cursor, cursor += firstGroupLen);

// Process remaining digit groups

**while** (cursor < len) {

**int** groupVal = parseInt(val, cursor, cursor += *digitsPerInt*[10]);

*destructiveMulAdd*(magnitude, *intRadix*[10], groupVal);

}

mag = *trustedStripLeadingZeroInts*(magnitude);

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

checkRange();

}

}

// Create an integer with the digits between the two indexes

// Assumes start < end. The result may be negative, but it

// is to be treated as an unsigned value.

**private** **int** parseInt(**char**[] source, **int** start, **int** end) {

**int** result = Character.*digit*(source[start++], 10);

**if** (result == -1)

**throw** **new** NumberFormatException(**new** String(source));

**for** (**int** index = start; index < end; index++) {

**int** nextVal = Character.*digit*(source[index], 10);

**if** (nextVal == -1)

**throw** **new** NumberFormatException(**new** String(source));

result = 10\*result + nextVal;

}

**return** result;

}

// bitsPerDigit in the given radix times 1024

// Rounded up to avoid underallocation.

**private** **static** **long** *bitsPerDigit*[] = { 0, 0,

1024, 1624, 2048, 2378, 2648, 2875, 3072, 3247, 3402, 3543, 3672,

3790, 3899, 4001, 4096, 4186, 4271, 4350, 4426, 4498, 4567, 4633,

4696, 4756, 4814, 4870, 4923, 4975, 5025, 5074, 5120, 5166, 5210,

5253, 5295};

// Multiply x array times word y in place, and add word z

**private** **static** **void** destructiveMulAdd(**int**[] x, **int** y, **int** z) {

// Perform the multiplication word by word

**long** ylong = y & ***LONG\_MASK***;

**long** zlong = z & ***LONG\_MASK***;

**int** len = x.length;

**long** product = 0;

**long** carry = 0;

**for** (**int** i = len-1; i >= 0; i--) {

product = ylong \* (x[i] & ***LONG\_MASK***) + carry;

x[i] = (**int**)product;

carry = product >>> 32;

}

// Perform the addition

**long** sum = (x[len-1] & ***LONG\_MASK***) + zlong;

x[len-1] = (**int**)sum;

carry = sum >>> 32;

**for** (**int** i = len-2; i >= 0; i--) {

sum = (x[i] & ***LONG\_MASK***) + carry;

x[i] = (**int**)sum;

carry = sum >>> 32;

}

}

/\*\*

\* Translates the decimal String representation of a BigInteger into a

\* BigInteger. The String representation consists of an optional minus

\* sign followed by a sequence of one or more decimal digits. The

\* character-to-digit mapping is provided by {@code Character.digit}.

\* The String may not contain any extraneous characters (whitespace, for

\* example).

\*

\* **@param** val decimal String representation of BigInteger.

\* **@throws** NumberFormatException {@code val} is not a valid representation

\* of a BigInteger.

\* **@see** Character#digit

\*/

**public** BigInteger(String val) {

**this**(val, 10);

}

/\*\*

\* Constructs a randomly generated BigInteger, uniformly distributed over

\* the range 0 to (2<sup>{@code numBits}</sup> - 1), inclusive.

\* The uniformity of the distribution assumes that a fair source of random

\* bits is provided in {@code rnd}. Note that this constructor always

\* constructs a non-negative BigInteger.

\*

\* **@param** numBits maximum bitLength of the new BigInteger.

\* **@param** rnd source of randomness to be used in computing the new

\* BigInteger.

\* **@throws** IllegalArgumentException {@code numBits} is negative.

\* **@see** #bitLength()

\*/

**public** BigInteger(**int** numBits, Random rnd) {

**this**(1, *randomBits*(numBits, rnd));

}

**private** **static** **byte**[] randomBits(**int** numBits, Random rnd) {

**if** (numBits < 0)

**throw** **new** IllegalArgumentException("numBits must be non-negative");

**int** numBytes = (**int**)(((**long**)numBits+7)/8); // avoid overflow

**byte**[] randomBits = **new** **byte**[numBytes];

// Generate random bytes and mask out any excess bits

**if** (numBytes > 0) {

rnd.nextBytes(randomBits);

**int** excessBits = 8\*numBytes - numBits;

randomBits[0] &= (1 << (8-excessBits)) - 1;

}

**return** randomBits;

}

/\*\*

\* Constructs a randomly generated positive BigInteger that is probably

\* prime, with the specified bitLength.

\*

\* <p>It is recommended that the {@link #probablePrime probablePrime}

\* method be used in preference to this constructor unless there

\* is a compelling need to specify a certainty.

\*

\* **@param** bitLength bitLength of the returned BigInteger.

\* **@param** certainty a measure of the uncertainty that the caller is

\* willing to tolerate. The probability that the new BigInteger

\* represents a prime number will exceed

\* (1 - 1/2<sup>{@code certainty}</sup>). The execution time of

\* this constructor is proportional to the value of this parameter.

\* **@param** rnd source of random bits used to select candidates to be

\* tested for primality.

\* **@throws** ArithmeticException {@code bitLength < 2} or {@code bitLength} is too large.

\* **@see** #bitLength()

\*/

**public** BigInteger(**int** bitLength, **int** certainty, Random rnd) {

BigInteger prime;

**if** (bitLength < 2)

**throw** **new** ArithmeticException("bitLength < 2");

prime = (bitLength < ***SMALL\_PRIME\_THRESHOLD***

? *smallPrime*(bitLength, certainty, rnd)

: *largePrime*(bitLength, certainty, rnd));

signum = 1;

mag = prime.mag;

}

// Minimum size in bits that the requested prime number has

// before we use the large prime number generating algorithms.

// The cutoff of 95 was chosen empirically for best performance.

**private** **static** **final** **int** ***SMALL\_PRIME\_THRESHOLD*** = 95;

// Certainty required to meet the spec of probablePrime

**private** **static** **final** **int** ***DEFAULT\_PRIME\_CERTAINTY*** = 100;

/\*\*

\* Returns a positive BigInteger that is probably prime, with the

\* specified bitLength. The probability that a BigInteger returned

\* by this method is composite does not exceed 2<sup>-100</sup>.

\*

\* **@param** bitLength bitLength of the returned BigInteger.

\* **@param** rnd source of random bits used to select candidates to be

\* tested for primality.

\* **@return** a BigInteger of {@code bitLength} bits that is probably prime

\* **@throws** ArithmeticException {@code bitLength < 2} or {@code bitLength} is too large.

\* **@see** #bitLength()

\* **@since** 1.4

\*/

**public** **static** BigInteger probablePrime(**int** bitLength, Random rnd) {

**if** (bitLength < 2)

**throw** **new** ArithmeticException("bitLength < 2");

**return** (bitLength < ***SMALL\_PRIME\_THRESHOLD*** ?

*smallPrime*(bitLength, ***DEFAULT\_PRIME\_CERTAINTY***, rnd) :

*largePrime*(bitLength, ***DEFAULT\_PRIME\_CERTAINTY***, rnd));

}

/\*\*

\* Find a random number of the specified bitLength that is probably prime.

\* This method is used for smaller primes, its performance degrades on

\* larger bitlengths.

\*

\* This method assumes bitLength > 1.

\*/

**private** **static** BigInteger smallPrime(**int** bitLength, **int** certainty, Random rnd) {

**int** magLen = (bitLength + 31) >>> 5;

**int** temp[] = **new** **int**[magLen];

**int** highBit = 1 << ((bitLength+31) & 0x1f); // High bit of high int

**int** highMask = (highBit << 1) - 1; // Bits to keep in high int

**while** (**true**) {

// Construct a candidate

**for** (**int** i=0; i < magLen; i++)

temp[i] = rnd.nextInt();

temp[0] = (temp[0] & highMask) | highBit; // Ensure exact length

**if** (bitLength > 2)

temp[magLen-1] |= 1; // Make odd if bitlen > 2

BigInteger p = **new** BigInteger(temp, 1);

// Do cheap "pre-test" if applicable

**if** (bitLength > 6) {

**long** r = p.remainder(***SMALL\_PRIME\_PRODUCT***).longValue();

**if** ((r%3==0) || (r%5==0) || (r%7==0) || (r%11==0) ||

(r%13==0) || (r%17==0) || (r%19==0) || (r%23==0) ||

(r%29==0) || (r%31==0) || (r%37==0) || (r%41==0))

**continue**; // Candidate is composite; try another

}

// All candidates of bitLength 2 and 3 are prime by this point

**if** (bitLength < 4)

**return** p;

// Do expensive test if we survive pre-test (or it's inapplicable)

**if** (p.primeToCertainty(certainty, rnd))

**return** p;

}

}

**private** **static** **final** BigInteger ***SMALL\_PRIME\_PRODUCT***

= *valueOf*(3L\*5\*7\*11\*13\*17\*19\*23\*29\*31\*37\*41);

/\*\*

\* Find a random number of the specified bitLength that is probably prime.

\* This method is more appropriate for larger bitlengths since it uses

\* a sieve to eliminate most composites before using a more expensive

\* test.

\*/

**private** **static** BigInteger largePrime(**int** bitLength, **int** certainty, Random rnd) {

BigInteger p;

p = **new** BigInteger(bitLength, rnd).setBit(bitLength-1);

p.mag[p.mag.length-1] &= 0xfffffffe;

// Use a sieve length likely to contain the next prime number

**int** searchLen = *getPrimeSearchLen*(bitLength);

BitSieve searchSieve = **new** BitSieve(p, searchLen);

BigInteger candidate = searchSieve.retrieve(p, certainty, rnd);

**while** ((candidate == **null**) || (candidate.bitLength() != bitLength)) {

p = p.add(BigInteger.*valueOf*(2\*searchLen));

**if** (p.bitLength() != bitLength)

p = **new** BigInteger(bitLength, rnd).setBit(bitLength-1);

p.mag[p.mag.length-1] &= 0xfffffffe;

searchSieve = **new** BitSieve(p, searchLen);

candidate = searchSieve.retrieve(p, certainty, rnd);

}

**return** candidate;

}

/\*\*

\* Returns the first integer greater than this {@code BigInteger} that

\* is probably prime. The probability that the number returned by this

\* method is composite does not exceed 2<sup>-100</sup>. This method will

\* never skip over a prime when searching: if it returns {@code p}, there

\* is no prime {@code q} such that {@code this < q < p}.

\*

\* **@return** the first integer greater than this {@code BigInteger} that

\* is probably prime.

\* **@throws** ArithmeticException {@code this < 0} or {@code this} is too large.

\* **@since** 1.5

\*/

**public** BigInteger nextProbablePrime() {

**if** (**this**.signum < 0)

**throw** **new** ArithmeticException("start < 0: " + **this**);

// Handle trivial cases

**if** ((**this**.signum == 0) || **this**.equals(***ONE***))

**return** ***TWO***;

BigInteger result = **this**.add(***ONE***);

// Fastpath for small numbers

**if** (result.bitLength() < ***SMALL\_PRIME\_THRESHOLD***) {

// Ensure an odd number

**if** (!result.testBit(0))

result = result.add(***ONE***);

**while** (**true**) {

// Do cheap "pre-test" if applicable

**if** (result.bitLength() > 6) {

**long** r = result.remainder(***SMALL\_PRIME\_PRODUCT***).longValue();

**if** ((r%3==0) || (r%5==0) || (r%7==0) || (r%11==0) ||

(r%13==0) || (r%17==0) || (r%19==0) || (r%23==0) ||

(r%29==0) || (r%31==0) || (r%37==0) || (r%41==0)) {

result = result.add(***TWO***);

**continue**; // Candidate is composite; try another

}

}

// All candidates of bitLength 2 and 3 are prime by this point

**if** (result.bitLength() < 4)

**return** result;

// The expensive test

**if** (result.primeToCertainty(***DEFAULT\_PRIME\_CERTAINTY***, **null**))

**return** result;

result = result.add(***TWO***);

}

}

// Start at previous even number

**if** (result.testBit(0))

result = result.subtract(***ONE***);

// Looking for the next large prime

**int** searchLen = *getPrimeSearchLen*(result.bitLength());

**while** (**true**) {

BitSieve searchSieve = **new** BitSieve(result, searchLen);

BigInteger candidate = searchSieve.retrieve(result,

***DEFAULT\_PRIME\_CERTAINTY***, **null**);

**if** (candidate != **null**)

**return** candidate;

result = result.add(BigInteger.*valueOf*(2 \* searchLen));

}

}

**private** **static** **int** getPrimeSearchLen(**int** bitLength) {

**if** (bitLength > ***PRIME\_SEARCH\_BIT\_LENGTH\_LIMIT*** + 1) {

**throw** **new** ArithmeticException("Prime search implementation restriction on bitLength");

}

**return** bitLength / 20 \* 64;

}

/\*\*

\* Returns {@code true} if this BigInteger is probably prime,

\* {@code false} if it's definitely composite.

\*

\* This method assumes bitLength > 2.

\*

\* **@param** certainty a measure of the uncertainty that the caller is

\* willing to tolerate: if the call returns {@code true}

\* the probability that this BigInteger is prime exceeds

\* {@code (1 - 1/2<sup>certainty</sup>)}. The execution time of

\* this method is proportional to the value of this parameter.

\* **@return** {@code true} if this BigInteger is probably prime,

\* {@code false} if it's definitely composite.

\*/

**boolean** primeToCertainty(**int** certainty, Random random) {

**int** rounds = 0;

**int** n = (Math.*min*(certainty, Integer.***MAX\_VALUE***-1)+1)/2;

// The relationship between the certainty and the number of rounds

// we perform is given in the draft standard ANSI X9.80, "PRIME

// NUMBER GENERATION, PRIMALITY TESTING, AND PRIMALITY CERTIFICATES".

**int** sizeInBits = **this**.bitLength();

**if** (sizeInBits < 100) {

rounds = 50;

rounds = n < rounds ? n : rounds;

**return** passesMillerRabin(rounds, random);

}

**if** (sizeInBits < 256) {

rounds = 27;

} **else** **if** (sizeInBits < 512) {

rounds = 15;

} **else** **if** (sizeInBits < 768) {

rounds = 8;

} **else** **if** (sizeInBits < 1024) {

rounds = 4;

} **else** {

rounds = 2;

}

rounds = n < rounds ? n : rounds;

**return** passesMillerRabin(rounds, random) && passesLucasLehmer();

}

/\*\*

\* Returns true iff this BigInteger is a Lucas-Lehmer probable prime.

\*

\* The following assumptions are made:

\* This BigInteger is a positive, odd number.

\*/

**private** **boolean** passesLucasLehmer() {

BigInteger thisPlusOne = **this**.add(***ONE***);

// Step 1

**int** d = 5;

**while** (*jacobiSymbol*(d, **this**) != -1) {

// 5, -7, 9, -11, ...

d = (d < 0) ? Math.*abs*(d)+2 : -(d+2);

}

// Step 2

BigInteger u = *lucasLehmerSequence*(d, thisPlusOne, **this**);

// Step 3

**return** u.mod(**this**).equals(***ZERO***);

}

/\*\*

\* Computes Jacobi(p,n).

\* Assumes n positive, odd, n>=3.

\*/

**private** **static** **int** jacobiSymbol(**int** p, BigInteger n) {

**if** (p == 0)

**return** 0;

// Algorithm and comments adapted from Colin Plumb's C library.

**int** j = 1;

**int** u = n.mag[n.mag.length-1];

// Make p positive

**if** (p < 0) {

p = -p;

**int** n8 = u & 7;

**if** ((n8 == 3) || (n8 == 7))

j = -j; // 3 (011) or 7 (111) mod 8

}

// Get rid of factors of 2 in p

**while** ((p & 3) == 0)

p >>= 2;

**if** ((p & 1) == 0) {

p >>= 1;

**if** (((u ^ (u>>1)) & 2) != 0)

j = -j; // 3 (011) or 5 (101) mod 8

}

**if** (p == 1)

**return** j;

// Then, apply quadratic reciprocity

**if** ((p & u & 2) != 0) // p = u = 3 (mod 4)?

j = -j;

// And reduce u mod p

u = n.mod(BigInteger.*valueOf*(p)).intValue();

// Now compute Jacobi(u,p), u < p

**while** (u != 0) {

**while** ((u & 3) == 0)

u >>= 2;

**if** ((u & 1) == 0) {

u >>= 1;

**if** (((p ^ (p>>1)) & 2) != 0)

j = -j; // 3 (011) or 5 (101) mod 8

}

**if** (u == 1)

**return** j;

// Now both u and p are odd, so use quadratic reciprocity

**assert** (u < p);

**int** t = u; u = p; p = t;

**if** ((u & p & 2) != 0) // u = p = 3 (mod 4)?

j = -j;

// Now u >= p, so it can be reduced

u %= p;

}

**return** 0;

}

**private** **static** BigInteger lucasLehmerSequence(**int** z, BigInteger k, BigInteger n) {

BigInteger d = BigInteger.*valueOf*(z);

BigInteger u = ***ONE***; BigInteger u2;

BigInteger v = ***ONE***; BigInteger v2;

**for** (**int** i=k.bitLength()-2; i >= 0; i--) {

u2 = u.multiply(v).mod(n);

v2 = v.square().add(d.multiply(u.square())).mod(n);

**if** (v2.testBit(0))

v2 = v2.subtract(n);

v2 = v2.shiftRight(1);

u = u2; v = v2;

**if** (k.testBit(i)) {

u2 = u.add(v).mod(n);

**if** (u2.testBit(0))

u2 = u2.subtract(n);

u2 = u2.shiftRight(1);

v2 = v.add(d.multiply(u)).mod(n);

**if** (v2.testBit(0))

v2 = v2.subtract(n);

v2 = v2.shiftRight(1);

u = u2; v = v2;

}

}

**return** u;

}

/\*\*

\* Returns true iff this BigInteger passes the specified number of

\* Miller-Rabin tests. This test is taken from the DSA spec (NIST FIPS

\* 186-2).

\*

\* The following assumptions are made:

\* This BigInteger is a positive, odd number greater than 2.

\* iterations<=50.

\*/

**private** **boolean** passesMillerRabin(**int** iterations, Random rnd) {

// Find a and m such that m is odd and this == 1 + 2\*\*a \* m

BigInteger thisMinusOne = **this**.subtract(***ONE***);

BigInteger m = thisMinusOne;

**int** a = m.getLowestSetBit();

m = m.shiftRight(a);

// Do the tests

**if** (rnd == **null**) {

rnd = ThreadLocalRandom.*current*();

}

**for** (**int** i=0; i < iterations; i++) {

// Generate a uniform random on (1, this)

BigInteger b;

**do** {

b = **new** BigInteger(**this**.bitLength(), rnd);

} **while** (b.compareTo(***ONE***) <= 0 || b.compareTo(**this**) >= 0);

**int** j = 0;

BigInteger z = b.modPow(m, **this**);

**while** (!((j == 0 && z.equals(***ONE***)) || z.equals(thisMinusOne))) {

**if** (j > 0 && z.equals(***ONE***) || ++j == a)

**return** **false**;

z = z.modPow(***TWO***, **this**);

}

}

**return** **true**;

}

/\*\*

\* This internal constructor differs from its public cousin

\* with the arguments reversed in two ways: it assumes that its

\* arguments are correct, and it doesn't copy the magnitude array.

\*/

BigInteger(**int**[] magnitude, **int** signum) {

**this**.signum = (magnitude.length == 0 ? 0 : signum);

**this**.mag = magnitude;

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

checkRange();

}

}

/\*\*

\* This private constructor is for internal use and assumes that its

\* arguments are correct.

\*/

**private** BigInteger(**byte**[] magnitude, **int** signum) {

**this**.signum = (magnitude.length == 0 ? 0 : signum);

**this**.mag = *stripLeadingZeroBytes*(magnitude);

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

checkRange();

}

}

/\*\*

\* Throws an {@code ArithmeticException} if the {@code BigInteger} would be

\* out of the supported range.

\*

\* **@throws** ArithmeticException if {@code this} exceeds the supported range.

\*/

**private** **void** checkRange() {

**if** (mag.length > ***MAX\_MAG\_LENGTH*** || mag.length == ***MAX\_MAG\_LENGTH*** && mag[0] < 0) {

*reportOverflow*();

}

}

**private** **static** **void** reportOverflow() {

**throw** **new** ArithmeticException("BigInteger would overflow supported range");

}

//Static Factory Methods

/\*\*

\* Returns a BigInteger whose value is equal to that of the

\* specified {@code long}. This "static factory method" is

\* provided in preference to a ({@code long}) constructor

\* because it allows for reuse of frequently used BigIntegers.

\*

\* **@param** val value of the BigInteger to return.

\* **@return** a BigInteger with the specified value.

\*/

**public** **static** BigInteger valueOf(**long** val) {

// If -MAX\_CONSTANT < val < MAX\_CONSTANT, return stashed constant

**if** (val == 0)

**return** ***ZERO***;

**if** (val > 0 && val <= ***MAX\_CONSTANT***)

**return** *posConst*[(**int**) val];

**else** **if** (val < 0 && val >= -***MAX\_CONSTANT***)

**return** *negConst*[(**int**) -val];

**return** **new** BigInteger(val);

}

/\*\*

\* Constructs a BigInteger with the specified value, which may not be zero.

\*/

**private** BigInteger(**long** val) {

**if** (val < 0) {

val = -val;

signum = -1;

} **else** {

signum = 1;

}

**int** highWord = (**int**)(val >>> 32);

**if** (highWord == 0) {

mag = **new** **int**[1];

mag[0] = (**int**)val;

} **else** {

mag = **new** **int**[2];

mag[0] = highWord;

mag[1] = (**int**)val;

}

}

/\*\*

\* Returns a BigInteger with the given two's complement representation.

\* Assumes that the input array will not be modified (the returned

\* BigInteger will reference the input array if feasible).

\*/

**private** **static** BigInteger valueOf(**int** val[]) {

**return** (val[0] > 0 ? **new** BigInteger(val, 1) : **new** BigInteger(val));

}

// Constants

/\*\*

\* Initialize static constant array when class is loaded.

\*/

**private** **final** **static** **int** ***MAX\_CONSTANT*** = 16;

**private** **static** BigInteger *posConst*[] = **new** BigInteger[***MAX\_CONSTANT***+1];

**private** **static** BigInteger *negConst*[] = **new** BigInteger[***MAX\_CONSTANT***+1];

/\*\*

\* The cache of powers of each radix. This allows us to not have to

\* recalculate powers of radix^(2^n) more than once. This speeds

\* Schoenhage recursive base conversion significantly.

\*/

**private** **static** **volatile** BigInteger[][] *powerCache*;

/\*\* The cache of logarithms of radices for base conversion. \*/

**private** **static** **final** **double**[] ***logCache***;

/\*\* The natural log of 2. This is used in computing cache indices. \*/

**private** **static** **final** **double** ***LOG\_TWO*** = Math.*log*(2.0);

**static** {

**for** (**int** i = 1; i <= ***MAX\_CONSTANT***; i++) {

**int**[] magnitude = **new** **int**[1];

magnitude[0] = i;

*posConst*[i] = **new** BigInteger(magnitude, 1);

*negConst*[i] = **new** BigInteger(magnitude, -1);

}

/\*

\* Initialize the cache of radix^(2^x) values used for base conversion

\* with just the very first value. Additional values will be created

\* on demand.

\*/

*powerCache* = **new** BigInteger[Character.***MAX\_RADIX***+1][];

***logCache*** = **new** **double**[Character.***MAX\_RADIX***+1];

**for** (**int** i=Character.***MIN\_RADIX***; i <= Character.***MAX\_RADIX***; i++) {

*powerCache*[i] = **new** BigInteger[] { BigInteger.*valueOf*(i) };

***logCache***[i] = Math.*log*(i);

}

}

/\*\*

\* The BigInteger constant zero.

\*

\* **@since** 1.2

\*/

**public** **static** **final** BigInteger ***ZERO*** = **new** BigInteger(**new** **int**[0], 0);

/\*\*

\* The BigInteger constant one.

\*

\* **@since** 1.2

\*/

**public** **static** **final** BigInteger ***ONE*** = *valueOf*(1);

/\*\*

\* The BigInteger constant two. (Not exported.)

\*/

**private** **static** **final** BigInteger ***TWO*** = *valueOf*(2);

/\*\*

\* The BigInteger constant -1. (Not exported.)

\*/

**private** **static** **final** BigInteger ***NEGATIVE\_ONE*** = *valueOf*(-1);

/\*\*

\* The BigInteger constant ten.

\*

\* **@since** 1.5

\*/

**public** **static** **final** BigInteger ***TEN*** = *valueOf*(10);

// Arithmetic Operations

/\*\*

\* Returns a BigInteger whose value is {@code (this + val)}.

\*

\* **@param** val value to be added to this BigInteger.

\* **@return** {@code this + val}

\*/

**public** BigInteger add(BigInteger val) {

**if** (val.signum == 0)

**return** **this**;

**if** (signum == 0)

**return** val;

**if** (val.signum == signum)

**return** **new** BigInteger(*add*(mag, val.mag), signum);

**int** cmp = compareMagnitude(val);

**if** (cmp == 0)

**return** ***ZERO***;

**int**[] resultMag = (cmp > 0 ? *subtract*(mag, val.mag)

: *subtract*(val.mag, mag));

resultMag = *trustedStripLeadingZeroInts*(resultMag);

**return** **new** BigInteger(resultMag, cmp == signum ? 1 : -1);

}

/\*\*

\* Package private methods used by BigDecimal code to add a BigInteger

\* with a long. Assumes val is not equal to INFLATED.

\*/

BigInteger add(**long** val) {

**if** (val == 0)

**return** **this**;

**if** (signum == 0)

**return** *valueOf*(val);

**if** (Long.*signum*(val) == signum)

**return** **new** BigInteger(*add*(mag, Math.*abs*(val)), signum);

**int** cmp = compareMagnitude(val);

**if** (cmp == 0)

**return** ***ZERO***;

**int**[] resultMag = (cmp > 0 ? *subtract*(mag, Math.*abs*(val)) : *subtract*(Math.*abs*(val), mag));

resultMag = *trustedStripLeadingZeroInts*(resultMag);

**return** **new** BigInteger(resultMag, cmp == signum ? 1 : -1);

}

/\*\*

\* Adds the contents of the int array x and long value val. This

\* method allocates a new int array to hold the answer and returns

\* a reference to that array. Assumes x.length &gt; 0 and val is

\* non-negative

\*/

**private** **static** **int**[] add(**int**[] x, **long** val) {

**int**[] y;

**long** sum = 0;

**int** xIndex = x.length;

**int**[] result;

**int** highWord = (**int**)(val >>> 32);

**if** (highWord == 0) {

result = **new** **int**[xIndex];

sum = (x[--xIndex] & ***LONG\_MASK***) + val;

result[xIndex] = (**int**)sum;

} **else** {

**if** (xIndex == 1) {

result = **new** **int**[2];

sum = val + (x[0] & ***LONG\_MASK***);

result[1] = (**int**)sum;

result[0] = (**int**)(sum >>> 32);

**return** result;

} **else** {

result = **new** **int**[xIndex];

sum = (x[--xIndex] & ***LONG\_MASK***) + (val & ***LONG\_MASK***);

result[xIndex] = (**int**)sum;

sum = (x[--xIndex] & ***LONG\_MASK***) + (highWord & ***LONG\_MASK***) + (sum >>> 32);

result[xIndex] = (**int**)sum;

}

}

// Copy remainder of longer number while carry propagation is required

**boolean** carry = (sum >>> 32 != 0);

**while** (xIndex > 0 && carry)

carry = ((result[--xIndex] = x[xIndex] + 1) == 0);

// Copy remainder of longer number

**while** (xIndex > 0)

result[--xIndex] = x[xIndex];

// Grow result if necessary

**if** (carry) {

**int** bigger[] = **new** **int**[result.length + 1];

System.*arraycopy*(result, 0, bigger, 1, result.length);

bigger[0] = 0x01;

**return** bigger;

}

**return** result;

}

/\*\*

\* Adds the contents of the int arrays x and y. This method allocates

\* a new int array to hold the answer and returns a reference to that

\* array.

\*/

**private** **static** **int**[] add(**int**[] x, **int**[] y) {

// If x is shorter, swap the two arrays

**if** (x.length < y.length) {

**int**[] tmp = x;

x = y;

y = tmp;

}

**int** xIndex = x.length;

**int** yIndex = y.length;

**int** result[] = **new** **int**[xIndex];

**long** sum = 0;

**if** (yIndex == 1) {

sum = (x[--xIndex] & ***LONG\_MASK***) + (y[0] & ***LONG\_MASK***) ;

result[xIndex] = (**int**)sum;

} **else** {

// Add common parts of both numbers

**while** (yIndex > 0) {

sum = (x[--xIndex] & ***LONG\_MASK***) +

(y[--yIndex] & ***LONG\_MASK***) + (sum >>> 32);

result[xIndex] = (**int**)sum;

}

}

// Copy remainder of longer number while carry propagation is required

**boolean** carry = (sum >>> 32 != 0);

**while** (xIndex > 0 && carry)

carry = ((result[--xIndex] = x[xIndex] + 1) == 0);

// Copy remainder of longer number

**while** (xIndex > 0)

result[--xIndex] = x[xIndex];

// Grow result if necessary

**if** (carry) {

**int** bigger[] = **new** **int**[result.length + 1];

System.*arraycopy*(result, 0, bigger, 1, result.length);

bigger[0] = 0x01;

**return** bigger;

}

**return** result;

}

**private** **static** **int**[] subtract(**long** val, **int**[] little) {

**int** highWord = (**int**)(val >>> 32);

**if** (highWord == 0) {

**int** result[] = **new** **int**[1];

result[0] = (**int**)(val - (little[0] & ***LONG\_MASK***));

**return** result;

} **else** {

**int** result[] = **new** **int**[2];

**if** (little.length == 1) {

**long** difference = ((**int**)val & ***LONG\_MASK***) - (little[0] & ***LONG\_MASK***);

result[1] = (**int**)difference;

// Subtract remainder of longer number while borrow propagates

**boolean** borrow = (difference >> 32 != 0);

**if** (borrow) {

result[0] = highWord - 1;

} **else** { // Copy remainder of longer number

result[0] = highWord;

}

**return** result;

} **else** { // little.length == 2

**long** difference = ((**int**)val & ***LONG\_MASK***) - (little[1] & ***LONG\_MASK***);

result[1] = (**int**)difference;

difference = (highWord & ***LONG\_MASK***) - (little[0] & ***LONG\_MASK***) + (difference >> 32);

result[0] = (**int**)difference;

**return** result;

}

}

}

/\*\*

\* Subtracts the contents of the second argument (val) from the

\* first (big). The first int array (big) must represent a larger number

\* than the second. This method allocates the space necessary to hold the

\* answer.

\* assumes val &gt;= 0

\*/

**private** **static** **int**[] subtract(**int**[] big, **long** val) {

**int** highWord = (**int**)(val >>> 32);

**int** bigIndex = big.length;

**int** result[] = **new** **int**[bigIndex];

**long** difference = 0;

**if** (highWord == 0) {

difference = (big[--bigIndex] & ***LONG\_MASK***) - val;

result[bigIndex] = (**int**)difference;

} **else** {

difference = (big[--bigIndex] & ***LONG\_MASK***) - (val & ***LONG\_MASK***);

result[bigIndex] = (**int**)difference;

difference = (big[--bigIndex] & ***LONG\_MASK***) - (highWord & ***LONG\_MASK***) + (difference >> 32);

result[bigIndex] = (**int**)difference;

}

// Subtract remainder of longer number while borrow propagates

**boolean** borrow = (difference >> 32 != 0);

**while** (bigIndex > 0 && borrow)

borrow = ((result[--bigIndex] = big[bigIndex] - 1) == -1);

// Copy remainder of longer number

**while** (bigIndex > 0)

result[--bigIndex] = big[bigIndex];

**return** result;

}

/\*\*

\* Returns a BigInteger whose value is {@code (this - val)}.

\*

\* **@param** val value to be subtracted from this BigInteger.

\* **@return** {@code this - val}

\*/

**public** BigInteger subtract(BigInteger val) {

**if** (val.signum == 0)

**return** **this**;

**if** (signum == 0)

**return** val.negate();

**if** (val.signum != signum)

**return** **new** BigInteger(*add*(mag, val.mag), signum);

**int** cmp = compareMagnitude(val);

**if** (cmp == 0)

**return** ***ZERO***;

**int**[] resultMag = (cmp > 0 ? *subtract*(mag, val.mag)

: *subtract*(val.mag, mag));

resultMag = *trustedStripLeadingZeroInts*(resultMag);

**return** **new** BigInteger(resultMag, cmp == signum ? 1 : -1);

}

/\*\*

\* Subtracts the contents of the second int arrays (little) from the

\* first (big). The first int array (big) must represent a larger number

\* than the second. This method allocates the space necessary to hold the

\* answer.

\*/

**private** **static** **int**[] subtract(**int**[] big, **int**[] little) {

**int** bigIndex = big.length;

**int** result[] = **new** **int**[bigIndex];

**int** littleIndex = little.length;

**long** difference = 0;

// Subtract common parts of both numbers

**while** (littleIndex > 0) {

difference = (big[--bigIndex] & ***LONG\_MASK***) -

(little[--littleIndex] & ***LONG\_MASK***) +

(difference >> 32);

result[bigIndex] = (**int**)difference;

}

// Subtract remainder of longer number while borrow propagates

**boolean** borrow = (difference >> 32 != 0);

**while** (bigIndex > 0 && borrow)

borrow = ((result[--bigIndex] = big[bigIndex] - 1) == -1);

// Copy remainder of longer number

**while** (bigIndex > 0)

result[--bigIndex] = big[bigIndex];

**return** result;

}

/\*\*

\* Returns a BigInteger whose value is {@code (this \* val)}.

\*

\* **@implNote** An implementation may offer better algorithmic

\* performance when {@code val == this}.

\*

\* **@param** val value to be multiplied by this BigInteger.

\* **@return** {@code this \* val}

\*/

**public** BigInteger multiply(BigInteger val) {

**if** (val.signum == 0 || signum == 0)

**return** ***ZERO***;

**int** xlen = mag.length;

**if** (val == **this** && xlen > ***MULTIPLY\_SQUARE\_THRESHOLD***) {

**return** square();

}

**int** ylen = val.mag.length;

**if** ((xlen < ***KARATSUBA\_THRESHOLD***) || (ylen < ***KARATSUBA\_THRESHOLD***)) {

**int** resultSign = signum == val.signum ? 1 : -1;

**if** (val.mag.length == 1) {

**return** *multiplyByInt*(mag,val.mag[0], resultSign);

}

**if** (mag.length == 1) {

**return** *multiplyByInt*(val.mag,mag[0], resultSign);

}

**int**[] result = *multiplyToLen*(mag, xlen,

val.mag, ylen, **null**);

result = *trustedStripLeadingZeroInts*(result);

**return** **new** BigInteger(result, resultSign);

} **else** {

**if** ((xlen < ***TOOM\_COOK\_THRESHOLD***) && (ylen < ***TOOM\_COOK\_THRESHOLD***)) {

**return** *multiplyKaratsuba*(**this**, val);

} **else** {

**return** *multiplyToomCook3*(**this**, val);

}

}

}

**private** **static** BigInteger multiplyByInt(**int**[] x, **int** y, **int** sign) {

**if** (Integer.*bitCount*(y) == 1) {

**return** **new** BigInteger(*shiftLeft*(x,Integer.*numberOfTrailingZeros*(y)), sign);

}

**int** xlen = x.length;

**int**[] rmag = **new** **int**[xlen + 1];

**long** carry = 0;

**long** yl = y & ***LONG\_MASK***;

**int** rstart = rmag.length - 1;

**for** (**int** i = xlen - 1; i >= 0; i--) {

**long** product = (x[i] & ***LONG\_MASK***) \* yl + carry;

rmag[rstart--] = (**int**)product;

carry = product >>> 32;

}

**if** (carry == 0L) {

rmag = java.util.Arrays.*copyOfRange*(rmag, 1, rmag.length);

} **else** {

rmag[rstart] = (**int**)carry;

}

**return** **new** BigInteger(rmag, sign);

}

/\*\*

\* Package private methods used by BigDecimal code to multiply a BigInteger

\* with a long. Assumes v is not equal to INFLATED.

\*/

BigInteger multiply(**long** v) {

**if** (v == 0 || signum == 0)

**return** ***ZERO***;

**if** (v == BigDecimal.***INFLATED***)

**return** multiply(BigInteger.*valueOf*(v));

**int** rsign = (v > 0 ? signum : -signum);

**if** (v < 0)

v = -v;

**long** dh = v >>> 32; // higher order bits

**long** dl = v & ***LONG\_MASK***; // lower order bits

**int** xlen = mag.length;

**int**[] value = mag;

**int**[] rmag = (dh == 0L) ? (**new** **int**[xlen + 1]) : (**new** **int**[xlen + 2]);

**long** carry = 0;

**int** rstart = rmag.length - 1;

**for** (**int** i = xlen - 1; i >= 0; i--) {

**long** product = (value[i] & ***LONG\_MASK***) \* dl + carry;

rmag[rstart--] = (**int**)product;

carry = product >>> 32;

}

rmag[rstart] = (**int**)carry;

**if** (dh != 0L) {

carry = 0;

rstart = rmag.length - 2;

**for** (**int** i = xlen - 1; i >= 0; i--) {

**long** product = (value[i] & ***LONG\_MASK***) \* dh +

(rmag[rstart] & ***LONG\_MASK***) + carry;

rmag[rstart--] = (**int**)product;

carry = product >>> 32;

}

rmag[0] = (**int**)carry;

}

**if** (carry == 0L)

rmag = java.util.Arrays.*copyOfRange*(rmag, 1, rmag.length);

**return** **new** BigInteger(rmag, rsign);

}

/\*\*

\* Multiplies int arrays x and y to the specified lengths and places

\* the result into z. There will be no leading zeros in the resultant array.

\*/

**private** **static** **int**[] multiplyToLen(**int**[] x, **int** xlen, **int**[] y, **int** ylen, **int**[] z) {

**int** xstart = xlen - 1;

**int** ystart = ylen - 1;

**if** (z == **null** || z.length < (xlen+ ylen))

z = **new** **int**[xlen+ylen];

**long** carry = 0;

**for** (**int** j=ystart, k=ystart+1+xstart; j >= 0; j--, k--) {

**long** product = (y[j] & ***LONG\_MASK***) \*

(x[xstart] & ***LONG\_MASK***) + carry;

z[k] = (**int**)product;

carry = product >>> 32;

}

z[xstart] = (**int**)carry;

**for** (**int** i = xstart-1; i >= 0; i--) {

carry = 0;

**for** (**int** j=ystart, k=ystart+1+i; j >= 0; j--, k--) {

**long** product = (y[j] & ***LONG\_MASK***) \*

(x[i] & ***LONG\_MASK***) +

(z[k] & ***LONG\_MASK***) + carry;

z[k] = (**int**)product;

carry = product >>> 32;

}

z[i] = (**int**)carry;

}

**return** z;

}

/\*\*

\* Multiplies two BigIntegers using the Karatsuba multiplication

\* algorithm. This is a recursive divide-and-conquer algorithm which is

\* more efficient for large numbers than what is commonly called the

\* "grade-school" algorithm used in multiplyToLen. If the numbers to be

\* multiplied have length n, the "grade-school" algorithm has an

\* asymptotic complexity of O(n^2). In contrast, the Karatsuba algorithm

\* has complexity of O(n^(log2(3))), or O(n^1.585). It achieves this

\* increased performance by doing 3 multiplies instead of 4 when

\* evaluating the product. As it has some overhead, should be used when

\* both numbers are larger than a certain threshold (found

\* experimentally).

\*

\* See: http://en.wikipedia.org/wiki/Karatsuba\_algorithm

\*/

**private** **static** BigInteger multiplyKaratsuba(BigInteger x, BigInteger y) {

**int** xlen = x.mag.length;

**int** ylen = y.mag.length;

// The number of ints in each half of the number.

**int** half = (Math.*max*(xlen, ylen)+1) / 2;

// xl and yl are the lower halves of x and y respectively,

// xh and yh are the upper halves.

BigInteger xl = x.getLower(half);

BigInteger xh = x.getUpper(half);

BigInteger yl = y.getLower(half);

BigInteger yh = y.getUpper(half);

BigInteger p1 = xh.multiply(yh); // p1 = xh\*yh

BigInteger p2 = xl.multiply(yl); // p2 = xl\*yl

// p3=(xh+xl)\*(yh+yl)

BigInteger p3 = xh.add(xl).multiply(yh.add(yl));

// result = p1 \* 2^(32\*2\*half) + (p3 - p1 - p2) \* 2^(32\*half) + p2

BigInteger result = p1.shiftLeft(32\*half).add(p3.subtract(p1).subtract(p2)).shiftLeft(32\*half).add(p2);

**if** (x.signum != y.signum) {

**return** result.negate();

} **else** {

**return** result;

}

}

/\*\*

\* Multiplies two BigIntegers using a 3-way Toom-Cook multiplication

\* algorithm. This is a recursive divide-and-conquer algorithm which is

\* more efficient for large numbers than what is commonly called the

\* "grade-school" algorithm used in multiplyToLen. If the numbers to be

\* multiplied have length n, the "grade-school" algorithm has an

\* asymptotic complexity of O(n^2). In contrast, 3-way Toom-Cook has a

\* complexity of about O(n^1.465). It achieves this increased asymptotic

\* performance by breaking each number into three parts and by doing 5

\* multiplies instead of 9 when evaluating the product. Due to overhead

\* (additions, shifts, and one division) in the Toom-Cook algorithm, it

\* should only be used when both numbers are larger than a certain

\* threshold (found experimentally). This threshold is generally larger

\* than that for Karatsuba multiplication, so this algorithm is generally

\* only used when numbers become significantly larger.

\*

\* The algorithm used is the "optimal" 3-way Toom-Cook algorithm outlined

\* by Marco Bodrato.

\*

\* See: http://bodrato.it/toom-cook/

\* http://bodrato.it/papers/#WAIFI2007

\*

\* "Towards Optimal Toom-Cook Multiplication for Univariate and

\* Multivariate Polynomials in Characteristic 2 and 0." by Marco BODRATO;

\* In C.Carlet and B.Sunar, Eds., "WAIFI'07 proceedings", p. 116-133,

\* LNCS #4547. Springer, Madrid, Spain, June 21-22, 2007.

\*

\*/

**private** **static** BigInteger multiplyToomCook3(BigInteger a, BigInteger b) {

**int** alen = a.mag.length;

**int** blen = b.mag.length;

**int** largest = Math.*max*(alen, blen);

// k is the size (in ints) of the lower-order slices.

**int** k = (largest+2)/3; // Equal to ceil(largest/3)

// r is the size (in ints) of the highest-order slice.

**int** r = largest - 2\*k;

// Obtain slices of the numbers. a2 and b2 are the most significant

// bits of the numbers a and b, and a0 and b0 the least significant.

BigInteger a0, a1, a2, b0, b1, b2;

a2 = a.getToomSlice(k, r, 0, largest);

a1 = a.getToomSlice(k, r, 1, largest);

a0 = a.getToomSlice(k, r, 2, largest);

b2 = b.getToomSlice(k, r, 0, largest);

b1 = b.getToomSlice(k, r, 1, largest);

b0 = b.getToomSlice(k, r, 2, largest);

BigInteger v0, v1, v2, vm1, vinf, t1, t2, tm1, da1, db1;

v0 = a0.multiply(b0);

da1 = a2.add(a0);

db1 = b2.add(b0);

vm1 = da1.subtract(a1).multiply(db1.subtract(b1));

da1 = da1.add(a1);

db1 = db1.add(b1);

v1 = da1.multiply(db1);

v2 = da1.add(a2).shiftLeft(1).subtract(a0).multiply(

db1.add(b2).shiftLeft(1).subtract(b0));

vinf = a2.multiply(b2);

// The algorithm requires two divisions by 2 and one by 3.

// All divisions are known to be exact, that is, they do not produce

// remainders, and all results are positive. The divisions by 2 are

// implemented as right shifts which are relatively efficient, leaving

// only an exact division by 3, which is done by a specialized

// linear-time algorithm.

t2 = v2.subtract(vm1).exactDivideBy3();

tm1 = v1.subtract(vm1).shiftRight(1);

t1 = v1.subtract(v0);

t2 = t2.subtract(t1).shiftRight(1);

t1 = t1.subtract(tm1).subtract(vinf);

t2 = t2.subtract(vinf.shiftLeft(1));

tm1 = tm1.subtract(t2);

// Number of bits to shift left.

**int** ss = k\*32;

BigInteger result = vinf.shiftLeft(ss).add(t2).shiftLeft(ss).add(t1).shiftLeft(ss).add(tm1).shiftLeft(ss).add(v0);

**if** (a.signum != b.signum) {

**return** result.negate();

} **else** {

**return** result;

}

}

/\*\*

\* Returns a slice of a BigInteger for use in Toom-Cook multiplication.

\*

\* **@param** lowerSize The size of the lower-order bit slices.

\* **@param** upperSize The size of the higher-order bit slices.

\* **@param** slice The index of which slice is requested, which must be a

\* number from 0 to size-1. Slice 0 is the highest-order bits, and slice

\* size-1 are the lowest-order bits. Slice 0 may be of different size than

\* the other slices.

\* **@param** fullsize The size of the larger integer array, used to align

\* slices to the appropriate position when multiplying different-sized

\* numbers.

\*/

**private** BigInteger getToomSlice(**int** lowerSize, **int** upperSize, **int** slice,

**int** fullsize) {

**int** start, end, sliceSize, len, offset;

len = mag.length;

offset = fullsize - len;

**if** (slice == 0) {

start = 0 - offset;

end = upperSize - 1 - offset;

} **else** {

start = upperSize + (slice-1)\*lowerSize - offset;

end = start + lowerSize - 1;

}

**if** (start < 0) {

start = 0;

}

**if** (end < 0) {

**return** ***ZERO***;

}

sliceSize = (end-start) + 1;

**if** (sliceSize <= 0) {

**return** ***ZERO***;

}

// While performing Toom-Cook, all slices are positive and

// the sign is adjusted when the final number is composed.

**if** (start == 0 && sliceSize >= len) {

**return** **this**.abs();

}

**int** intSlice[] = **new** **int**[sliceSize];

System.*arraycopy*(mag, start, intSlice, 0, sliceSize);

**return** **new** BigInteger(*trustedStripLeadingZeroInts*(intSlice), 1);

}

/\*\*

\* Does an exact division (that is, the remainder is known to be zero)

\* of the specified number by 3. This is used in Toom-Cook

\* multiplication. This is an efficient algorithm that runs in linear

\* time. If the argument is not exactly divisible by 3, results are

\* undefined. Note that this is expected to be called with positive

\* arguments only.

\*/

**private** BigInteger exactDivideBy3() {

**int** len = mag.length;

**int**[] result = **new** **int**[len];

**long** x, w, q, borrow;

borrow = 0L;

**for** (**int** i=len-1; i >= 0; i--) {

x = (mag[i] & ***LONG\_MASK***);

w = x - borrow;

**if** (borrow > x) { // Did we make the number go negative?

borrow = 1L;

} **else** {

borrow = 0L;

}

// 0xAAAAAAAB is the modular inverse of 3 (mod 2^32). Thus,

// the effect of this is to divide by 3 (mod 2^32).

// This is much faster than division on most architectures.

q = (w \* 0xAAAAAAABL) & ***LONG\_MASK***;

result[i] = (**int**) q;

// Now check the borrow. The second check can of course be

// eliminated if the first fails.

**if** (q >= 0x55555556L) {

borrow++;

**if** (q >= 0xAAAAAAABL)

borrow++;

}

}

result = *trustedStripLeadingZeroInts*(result);

**return** **new** BigInteger(result, signum);

}

/\*\*

\* Returns a new BigInteger representing n lower ints of the number.

\* This is used by Karatsuba multiplication and Karatsuba squaring.

\*/

**private** BigInteger getLower(**int** n) {

**int** len = mag.length;

**if** (len <= n) {

**return** abs();

}

**int** lowerInts[] = **new** **int**[n];

System.*arraycopy*(mag, len-n, lowerInts, 0, n);

**return** **new** BigInteger(*trustedStripLeadingZeroInts*(lowerInts), 1);

}

/\*\*

\* Returns a new BigInteger representing mag.length-n upper

\* ints of the number. This is used by Karatsuba multiplication and

\* Karatsuba squaring.

\*/

**private** BigInteger getUpper(**int** n) {

**int** len = mag.length;

**if** (len <= n) {

**return** ***ZERO***;

}

**int** upperLen = len - n;

**int** upperInts[] = **new** **int**[upperLen];

System.*arraycopy*(mag, 0, upperInts, 0, upperLen);

**return** **new** BigInteger(*trustedStripLeadingZeroInts*(upperInts), 1);

}

// Squaring

/\*\*

\* Returns a BigInteger whose value is {@code (this<sup>2</sup>)}.

\*

\* **@return** {@code this<sup>2</sup>}

\*/

**private** BigInteger square() {

**if** (signum == 0) {

**return** ***ZERO***;

}

**int** len = mag.length;

**if** (len < ***KARATSUBA\_SQUARE\_THRESHOLD***) {

**int**[] z = *squareToLen*(mag, len, **null**);

**return** **new** BigInteger(*trustedStripLeadingZeroInts*(z), 1);

} **else** {

**if** (len < ***TOOM\_COOK\_SQUARE\_THRESHOLD***) {

**return** squareKaratsuba();

} **else** {

**return** squareToomCook3();

}

}

}

/\*\*

\* Squares the contents of the int array x. The result is placed into the

\* int array z. The contents of x are not changed.

\*/

**private** **static** **final** **int**[] squareToLen(**int**[] x, **int** len, **int**[] z) {

**int** zlen = len << 1;

**if** (z == **null** || z.length < zlen)

z = **new** **int**[zlen];

// Execute checks before calling intrinsified method.

*implSquareToLenChecks*(x, len, z, zlen);

**return** *implSquareToLen*(x, len, z, zlen);

}

/\*\*

\* Parameters validation.

\*/

**private** **static** **void** implSquareToLenChecks(**int**[] x, **int** len, **int**[] z, **int** zlen) **throws** RuntimeException {

**if** (len < 1) {

**throw** **new** IllegalArgumentException("invalid input length: " + len);

}

**if** (len > x.length) {

**throw** **new** IllegalArgumentException("input length out of bound: " +

len + " > " + x.length);

}

**if** (len \* 2 > z.length) {

**throw** **new** IllegalArgumentException("input length out of bound: " +

(len \* 2) + " > " + z.length);

}

**if** (zlen < 1) {

**throw** **new** IllegalArgumentException("invalid input length: " + zlen);

}

**if** (zlen > z.length) {

**throw** **new** IllegalArgumentException("input length out of bound: " +

len + " > " + z.length);

}

}

/\*\*

\* Java Runtime may use intrinsic for this method.

\*/

**private** **static** **final** **int**[] implSquareToLen(**int**[] x, **int** len, **int**[] z, **int** zlen) {

/\*

\* The algorithm used here is adapted from Colin Plumb's C library.

\* Technique: Consider the partial products in the multiplication

\* of "abcde" by itself:

\*

\* a b c d e

\* \* a b c d e

\* ==================

\* ae be ce de ee

\* ad bd cd dd de

\* ac bc cc cd ce

\* ab bb bc bd be

\* aa ab ac ad ae

\*

\* Note that everything above the main diagonal:

\* ae be ce de = (abcd) \* e

\* ad bd cd = (abc) \* d

\* ac bc = (ab) \* c

\* ab = (a) \* b

\*

\* is a copy of everything below the main diagonal:

\* de

\* cd ce

\* bc bd be

\* ab ac ad ae

\*

\* Thus, the sum is 2 \* (off the diagonal) + diagonal.

\*

\* This is accumulated beginning with the diagonal (which

\* consist of the squares of the digits of the input), which is then

\* divided by two, the off-diagonal added, and multiplied by two

\* again. The low bit is simply a copy of the low bit of the

\* input, so it doesn't need special care.

\*/

// Store the squares, right shifted one bit (i.e., divided by 2)

**int** lastProductLowWord = 0;

**for** (**int** j=0, i=0; j < len; j++) {

**long** piece = (x[j] & ***LONG\_MASK***);

**long** product = piece \* piece;

z[i++] = (lastProductLowWord << 31) | (**int**)(product >>> 33);

z[i++] = (**int**)(product >>> 1);

lastProductLowWord = (**int**)product;

}

// Add in off-diagonal sums

**for** (**int** i=len, offset=1; i > 0; i--, offset+=2) {

**int** t = x[i-1];

t = *mulAdd*(z, x, offset, i-1, t);

*addOne*(z, offset-1, i, t);

}

// Shift back up and set low bit

*primitiveLeftShift*(z, zlen, 1);

z[zlen-1] |= x[len-1] & 1;

**return** z;

}

/\*\*

\* Squares a BigInteger using the Karatsuba squaring algorithm. It should

\* be used when both numbers are larger than a certain threshold (found

\* experimentally). It is a recursive divide-and-conquer algorithm that

\* has better asymptotic performance than the algorithm used in

\* squareToLen.

\*/

**private** BigInteger squareKaratsuba() {

**int** half = (mag.length+1) / 2;

BigInteger xl = getLower(half);

BigInteger xh = getUpper(half);

BigInteger xhs = xh.square(); // xhs = xh^2

BigInteger xls = xl.square(); // xls = xl^2

// xh^2 << 64 + (((xl+xh)^2 - (xh^2 + xl^2)) << 32) + xl^2

**return** xhs.shiftLeft(half\*32).add(xl.add(xh).square().subtract(xhs.add(xls))).shiftLeft(half\*32).add(xls);

}

/\*\*

\* Squares a BigInteger using the 3-way Toom-Cook squaring algorithm. It

\* should be used when both numbers are larger than a certain threshold

\* (found experimentally). It is a recursive divide-and-conquer algorithm

\* that has better asymptotic performance than the algorithm used in

\* squareToLen or squareKaratsuba.

\*/

**private** BigInteger squareToomCook3() {

**int** len = mag.length;

// k is the size (in ints) of the lower-order slices.

**int** k = (len+2)/3; // Equal to ceil(largest/3)

// r is the size (in ints) of the highest-order slice.

**int** r = len - 2\*k;

// Obtain slices of the numbers. a2 is the most significant

// bits of the number, and a0 the least significant.

BigInteger a0, a1, a2;

a2 = getToomSlice(k, r, 0, len);

a1 = getToomSlice(k, r, 1, len);

a0 = getToomSlice(k, r, 2, len);

BigInteger v0, v1, v2, vm1, vinf, t1, t2, tm1, da1;

v0 = a0.square();

da1 = a2.add(a0);

vm1 = da1.subtract(a1).square();

da1 = da1.add(a1);

v1 = da1.square();

vinf = a2.square();

v2 = da1.add(a2).shiftLeft(1).subtract(a0).square();

// The algorithm requires two divisions by 2 and one by 3.

// All divisions are known to be exact, that is, they do not produce

// remainders, and all results are positive. The divisions by 2 are

// implemented as right shifts which are relatively efficient, leaving

// only a division by 3.

// The division by 3 is done by an optimized algorithm for this case.

t2 = v2.subtract(vm1).exactDivideBy3();

tm1 = v1.subtract(vm1).shiftRight(1);

t1 = v1.subtract(v0);

t2 = t2.subtract(t1).shiftRight(1);

t1 = t1.subtract(tm1).subtract(vinf);

t2 = t2.subtract(vinf.shiftLeft(1));

tm1 = tm1.subtract(t2);

// Number of bits to shift left.

**int** ss = k\*32;

**return** vinf.shiftLeft(ss).add(t2).shiftLeft(ss).add(t1).shiftLeft(ss).add(tm1).shiftLeft(ss).add(v0);

}

// Division

/\*\*

\* Returns a BigInteger whose value is {@code (this / val)}.

\*

\* **@param** val value by which this BigInteger is to be divided.

\* **@return** {@code this / val}

\* **@throws** ArithmeticException if {@code val} is zero.

\*/

**public** BigInteger divide(BigInteger val) {

**if** (val.mag.length < ***BURNIKEL\_ZIEGLER\_THRESHOLD*** ||

mag.length - val.mag.length < ***BURNIKEL\_ZIEGLER\_OFFSET***) {

**return** divideKnuth(val);

} **else** {

**return** divideBurnikelZiegler(val);

}

}

/\*\*

\* Returns a BigInteger whose value is {@code (this / val)} using an O(n^2) algorithm from Knuth.

\*

\* **@param** val value by which this BigInteger is to be divided.

\* **@return** {@code this / val}

\* **@throws** ArithmeticException if {@code val} is zero.

\* **@see** MutableBigInteger#divideKnuth(MutableBigInteger, MutableBigInteger, boolean)

\*/

**private** BigInteger divideKnuth(BigInteger val) {

MutableBigInteger q = **new** MutableBigInteger(),

a = **new** MutableBigInteger(**this**.mag),

b = **new** MutableBigInteger(val.mag);

a.divideKnuth(b, q, **false**);

**return** q.toBigInteger(**this**.signum \* val.signum);

}

/\*\*

\* Returns an array of two BigIntegers containing {@code (this / val)}

\* followed by {@code (this % val)}.

\*

\* **@param** val value by which this BigInteger is to be divided, and the

\* remainder computed.

\* **@return** an array of two BigIntegers: the quotient {@code (this / val)}

\* is the initial element, and the remainder {@code (this % val)}

\* is the final element.

\* **@throws** ArithmeticException if {@code val} is zero.

\*/

**public** BigInteger[] divideAndRemainder(BigInteger val) {

**if** (val.mag.length < ***BURNIKEL\_ZIEGLER\_THRESHOLD*** ||

mag.length - val.mag.length < ***BURNIKEL\_ZIEGLER\_OFFSET***) {

**return** divideAndRemainderKnuth(val);

} **else** {

**return** divideAndRemainderBurnikelZiegler(val);

}

}

/\*\* Long division \*/

**private** BigInteger[] divideAndRemainderKnuth(BigInteger val) {

BigInteger[] result = **new** BigInteger[2];

MutableBigInteger q = **new** MutableBigInteger(),

a = **new** MutableBigInteger(**this**.mag),

b = **new** MutableBigInteger(val.mag);

MutableBigInteger r = a.divideKnuth(b, q);

result[0] = q.toBigInteger(**this**.signum == val.signum ? 1 : -1);

result[1] = r.toBigInteger(**this**.signum);

**return** result;

}

/\*\*

\* Returns a BigInteger whose value is {@code (this % val)}.

\*

\* **@param** val value by which this BigInteger is to be divided, and the

\* remainder computed.

\* **@return** {@code this % val}

\* **@throws** ArithmeticException if {@code val} is zero.

\*/

**public** BigInteger remainder(BigInteger val) {

**if** (val.mag.length < ***BURNIKEL\_ZIEGLER\_THRESHOLD*** ||

mag.length - val.mag.length < ***BURNIKEL\_ZIEGLER\_OFFSET***) {

**return** remainderKnuth(val);

} **else** {

**return** remainderBurnikelZiegler(val);

}

}

/\*\* Long division \*/

**private** BigInteger remainderKnuth(BigInteger val) {

MutableBigInteger q = **new** MutableBigInteger(),

a = **new** MutableBigInteger(**this**.mag),

b = **new** MutableBigInteger(val.mag);

**return** a.divideKnuth(b, q).toBigInteger(**this**.signum);

}

/\*\*

\* Calculates {@code this / val} using the Burnikel-Ziegler algorithm.

\* **@param** val the divisor

\* **@return** {@code this / val}

\*/

**private** BigInteger divideBurnikelZiegler(BigInteger val) {

**return** divideAndRemainderBurnikelZiegler(val)[0];

}

/\*\*

\* Calculates {@code this % val} using the Burnikel-Ziegler algorithm.

\* **@param** val the divisor

\* **@return** {@code this % val}

\*/

**private** BigInteger remainderBurnikelZiegler(BigInteger val) {

**return** divideAndRemainderBurnikelZiegler(val)[1];

}

/\*\*

\* Computes {@code this / val} and {@code this % val} using the

\* Burnikel-Ziegler algorithm.

\* **@param** val the divisor

\* **@return** an array containing the quotient and remainder

\*/

**private** BigInteger[] divideAndRemainderBurnikelZiegler(BigInteger val) {

MutableBigInteger q = **new** MutableBigInteger();

MutableBigInteger r = **new** MutableBigInteger(**this**).divideAndRemainderBurnikelZiegler(**new** MutableBigInteger(val), q);

BigInteger qBigInt = q.isZero() ? ***ZERO*** : q.toBigInteger(signum\*val.signum);

BigInteger rBigInt = r.isZero() ? ***ZERO*** : r.toBigInteger(signum);

**return** **new** BigInteger[] {qBigInt, rBigInt};

}

/\*\*

\* Returns a BigInteger whose value is <tt>(this<sup>exponent</sup>)</tt>.

\* Note that {@code exponent} is an integer rather than a BigInteger.

\*

\* **@param** exponent exponent to which this BigInteger is to be raised.

\* **@return** <tt>this<sup>exponent</sup></tt>

\* **@throws** ArithmeticException {@code exponent} is negative. (This would

\* cause the operation to yield a non-integer value.)

\*/

**public** BigInteger pow(**int** exponent) {

**if** (exponent < 0) {

**throw** **new** ArithmeticException("Negative exponent");

}

**if** (signum == 0) {

**return** (exponent == 0 ? ***ONE*** : **this**);

}

BigInteger partToSquare = **this**.abs();

// Factor out powers of two from the base, as the exponentiation of

// these can be done by left shifts only.

// The remaining part can then be exponentiated faster. The

// powers of two will be multiplied back at the end.

**int** powersOfTwo = partToSquare.getLowestSetBit();

**long** bitsToShift = (**long**)powersOfTwo \* exponent;

**if** (bitsToShift > Integer.***MAX\_VALUE***) {

*reportOverflow*();

}

**int** remainingBits;

// Factor the powers of two out quickly by shifting right, if needed.

**if** (powersOfTwo > 0) {

partToSquare = partToSquare.shiftRight(powersOfTwo);

remainingBits = partToSquare.bitLength();

**if** (remainingBits == 1) { // Nothing left but +/- 1?

**if** (signum < 0 && (exponent&1) == 1) {

**return** ***NEGATIVE\_ONE***.shiftLeft(powersOfTwo\*exponent);

} **else** {

**return** ***ONE***.shiftLeft(powersOfTwo\*exponent);

}

}

} **else** {

remainingBits = partToSquare.bitLength();

**if** (remainingBits == 1) { // Nothing left but +/- 1?

**if** (signum < 0 && (exponent&1) == 1) {

**return** ***NEGATIVE\_ONE***;

} **else** {

**return** ***ONE***;

}

}

}

// This is a quick way to approximate the size of the result,

// similar to doing log2[n] \* exponent. This will give an upper bound

// of how big the result can be, and which algorithm to use.

**long** scaleFactor = (**long**)remainingBits \* exponent;

// Use slightly different algorithms for small and large operands.

// See if the result will safely fit into a long. (Largest 2^63-1)

**if** (partToSquare.mag.length == 1 && scaleFactor <= 62) {

// Small number algorithm. Everything fits into a long.

**int** newSign = (signum <0 && (exponent&1) == 1 ? -1 : 1);

**long** result = 1;

**long** baseToPow2 = partToSquare.mag[0] & ***LONG\_MASK***;

**int** workingExponent = exponent;

// Perform exponentiation using repeated squaring trick

**while** (workingExponent != 0) {

**if** ((workingExponent & 1) == 1) {

result = result \* baseToPow2;

}

**if** ((workingExponent >>>= 1) != 0) {

baseToPow2 = baseToPow2 \* baseToPow2;

}

}

// Multiply back the powers of two (quickly, by shifting left)

**if** (powersOfTwo > 0) {

**if** (bitsToShift + scaleFactor <= 62) { // Fits in long?

**return** *valueOf*((result << bitsToShift) \* newSign);

} **else** {

**return** *valueOf*(result\*newSign).shiftLeft((**int**) bitsToShift);

}

}

**else** {

**return** *valueOf*(result\*newSign);

}

} **else** {

// Large number algorithm. This is basically identical to

// the algorithm above, but calls multiply() and square()

// which may use more efficient algorithms for large numbers.

BigInteger answer = ***ONE***;

**int** workingExponent = exponent;

// Perform exponentiation using repeated squaring trick

**while** (workingExponent != 0) {

**if** ((workingExponent & 1) == 1) {

answer = answer.multiply(partToSquare);

}

**if** ((workingExponent >>>= 1) != 0) {

partToSquare = partToSquare.square();

}

}

// Multiply back the (exponentiated) powers of two (quickly,

// by shifting left)

**if** (powersOfTwo > 0) {

answer = answer.shiftLeft(powersOfTwo\*exponent);

}

**if** (signum < 0 && (exponent&1) == 1) {

**return** answer.negate();

} **else** {

**return** answer;

}

}

}

/\*\*

\* Returns a BigInteger whose value is the greatest common divisor of

\* {@code abs(this)} and {@code abs(val)}. Returns 0 if

\* {@code this == 0 && val == 0}.

\*

\* **@param** val value with which the GCD is to be computed.

\* **@return** {@code GCD(abs(this), abs(val))}

\*/

**public** BigInteger gcd(BigInteger val) {

**if** (val.signum == 0)

**return** **this**.abs();

**else** **if** (**this**.signum == 0)

**return** val.abs();

MutableBigInteger a = **new** MutableBigInteger(**this**);

MutableBigInteger b = **new** MutableBigInteger(val);

MutableBigInteger result = a.hybridGCD(b);

**return** result.toBigInteger(1);

}

/\*\*

\* Package private method to return bit length for an integer.

\*/

**static** **int** bitLengthForInt(**int** n) {

**return** 32 - Integer.*numberOfLeadingZeros*(n);

}

/\*\*

\* Left shift int array a up to len by n bits. Returns the array that

\* results from the shift since space may have to be reallocated.

\*/

**private** **static** **int**[] leftShift(**int**[] a, **int** len, **int** n) {

**int** nInts = n >>> 5;

**int** nBits = n&0x1F;

**int** bitsInHighWord = *bitLengthForInt*(a[0]);

// If shift can be done without recopy, do so

**if** (n <= (32-bitsInHighWord)) {

*primitiveLeftShift*(a, len, nBits);

**return** a;

} **else** { // Array must be resized

**if** (nBits <= (32-bitsInHighWord)) {

**int** result[] = **new** **int**[nInts+len];

System.*arraycopy*(a, 0, result, 0, len);

*primitiveLeftShift*(result, result.length, nBits);

**return** result;

} **else** {

**int** result[] = **new** **int**[nInts+len+1];

System.*arraycopy*(a, 0, result, 0, len);

*primitiveRightShift*(result, result.length, 32 - nBits);

**return** result;

}

}

}

// shifts a up to len right n bits assumes no leading zeros, 0<n<32

**static** **void** primitiveRightShift(**int**[] a, **int** len, **int** n) {

**int** n2 = 32 - n;

**for** (**int** i=len-1, c=a[i]; i > 0; i--) {

**int** b = c;

c = a[i-1];

a[i] = (c << n2) | (b >>> n);

}

a[0] >>>= n;

}

// shifts a up to len left n bits assumes no leading zeros, 0<=n<32

**static** **void** primitiveLeftShift(**int**[] a, **int** len, **int** n) {

**if** (len == 0 || n == 0)

**return**;

**int** n2 = 32 - n;

**for** (**int** i=0, c=a[i], m=i+len-1; i < m; i++) {

**int** b = c;

c = a[i+1];

a[i] = (b << n) | (c >>> n2);

}

a[len-1] <<= n;

}

/\*\*

\* Calculate bitlength of contents of the first len elements an int array,

\* assuming there are no leading zero ints.

\*/

**private** **static** **int** bitLength(**int**[] val, **int** len) {

**if** (len == 0)

**return** 0;

**return** ((len - 1) << 5) + *bitLengthForInt*(val[0]);

}

/\*\*

\* Returns a BigInteger whose value is the absolute value of this

\* BigInteger.

\*

\* **@return** {@code abs(this)}

\*/

**public** BigInteger abs() {

**return** (signum >= 0 ? **this** : **this**.negate());

}

/\*\*

\* Returns a BigInteger whose value is {@code (-this)}.

\*

\* **@return** {@code -this}

\*/

**public** BigInteger negate() {

**return** **new** BigInteger(**this**.mag, -**this**.signum);

}

/\*\*

\* Returns the signum function of this BigInteger.

\*

\* **@return** -1, 0 or 1 as the value of this BigInteger is negative, zero or

\* positive.

\*/

**public** **int** signum() {

**return** **this**.signum;

}

// Modular Arithmetic Operations

/\*\*

\* Returns a BigInteger whose value is {@code (this mod m}). This method

\* differs from {@code remainder} in that it always returns a

\* <i>non-negative</i> BigInteger.

\*

\* **@param** m the modulus.

\* **@return** {@code this mod m}

\* **@throws** ArithmeticException {@code m} &le; 0

\* **@see** #remainder

\*/

**public** BigInteger mod(BigInteger m) {

**if** (m.signum <= 0)

**throw** **new** ArithmeticException("BigInteger: modulus not positive");

BigInteger result = **this**.remainder(m);

**return** (result.signum >= 0 ? result : result.add(m));

}

/\*\*

\* Returns a BigInteger whose value is

\* <tt>(this<sup>exponent</sup> mod m)</tt>. (Unlike {@code pow}, this

\* method permits negative exponents.)

\*

\* **@param** exponent the exponent.

\* **@param** m the modulus.

\* **@return** <tt>this<sup>exponent</sup> mod m</tt>

\* **@throws** ArithmeticException {@code m} &le; 0 or the exponent is

\* negative and this BigInteger is not <i>relatively

\* prime</i> to {@code m}.

\* **@see** #modInverse

\*/

**public** BigInteger modPow(BigInteger exponent, BigInteger m) {

**if** (m.signum <= 0)

**throw** **new** ArithmeticException("BigInteger: modulus not positive");

// Trivial cases

**if** (exponent.signum == 0)

**return** (m.equals(***ONE***) ? ***ZERO*** : ***ONE***);

**if** (**this**.equals(***ONE***))

**return** (m.equals(***ONE***) ? ***ZERO*** : ***ONE***);

**if** (**this**.equals(***ZERO***) && exponent.signum >= 0)

**return** ***ZERO***;

**if** (**this**.equals(*negConst*[1]) && (!exponent.testBit(0)))

**return** (m.equals(***ONE***) ? ***ZERO*** : ***ONE***);

**boolean** invertResult;

**if** ((invertResult = (exponent.signum < 0)))

exponent = exponent.negate();

BigInteger base = (**this**.signum < 0 || **this**.compareTo(m) >= 0

? **this**.mod(m) : **this**);

BigInteger result;

**if** (m.testBit(0)) { // odd modulus

result = base.oddModPow(exponent, m);

} **else** {

/\*

\* Even modulus. Tear it into an "odd part" (m1) and power of two

\* (m2), exponentiate mod m1, manually exponentiate mod m2, and

\* use Chinese Remainder Theorem to combine results.

\*/

// Tear m apart into odd part (m1) and power of 2 (m2)

**int** p = m.getLowestSetBit(); // Max pow of 2 that divides m

BigInteger m1 = m.shiftRight(p); // m/2\*\*p

BigInteger m2 = ***ONE***.shiftLeft(p); // 2\*\*p

// Calculate new base from m1

BigInteger base2 = (**this**.signum < 0 || **this**.compareTo(m1) >= 0

? **this**.mod(m1) : **this**);

// Caculate (base \*\* exponent) mod m1.

BigInteger a1 = (m1.equals(***ONE***) ? ***ZERO*** :

base2.oddModPow(exponent, m1));

// Calculate (this \*\* exponent) mod m2

BigInteger a2 = base.modPow2(exponent, p);

// Combine results using Chinese Remainder Theorem

BigInteger y1 = m2.modInverse(m1);

BigInteger y2 = m1.modInverse(m2);

**if** (m.mag.length < ***MAX\_MAG\_LENGTH*** / 2) {

result = a1.multiply(m2).multiply(y1).add(a2.multiply(m1).multiply(y2)).mod(m);

} **else** {

MutableBigInteger t1 = **new** MutableBigInteger();

**new** MutableBigInteger(a1.multiply(m2)).multiply(**new** MutableBigInteger(y1), t1);

MutableBigInteger t2 = **new** MutableBigInteger();

**new** MutableBigInteger(a2.multiply(m1)).multiply(**new** MutableBigInteger(y2), t2);

t1.add(t2);

MutableBigInteger q = **new** MutableBigInteger();

result = t1.divide(**new** MutableBigInteger(m), q).toBigInteger();

}

}

**return** (invertResult ? result.modInverse(m) : result);

}

// Montgomery multiplication. These are wrappers for

// implMontgomeryXX routines which are expected to be replaced by

// virtual machine intrinsics. We don't use the intrinsics for

// very large operands: MONTGOMERY\_INTRINSIC\_THRESHOLD should be

// larger than any reasonable crypto key.

**private** **static** **int**[] montgomeryMultiply(**int**[] a, **int**[] b, **int**[] n, **int** len, **long** inv,

**int**[] product) {

*implMontgomeryMultiplyChecks*(a, b, n, len, product);

**if** (len > ***MONTGOMERY\_INTRINSIC\_THRESHOLD***) {

// Very long argument: do not use an intrinsic

product = *multiplyToLen*(a, len, b, len, product);

**return** *montReduce*(product, n, len, (**int**)inv);

} **else** {

**return** *implMontgomeryMultiply*(a, b, n, len, inv, *materialize*(product, len));

}

}

**private** **static** **int**[] montgomerySquare(**int**[] a, **int**[] n, **int** len, **long** inv,

**int**[] product) {

*implMontgomeryMultiplyChecks*(a, a, n, len, product);

**if** (len > ***MONTGOMERY\_INTRINSIC\_THRESHOLD***) {

// Very long argument: do not use an intrinsic

product = *squareToLen*(a, len, product);

**return** *montReduce*(product, n, len, (**int**)inv);

} **else** {

**return** *implMontgomerySquare*(a, n, len, inv, *materialize*(product, len));

}

}

// Range-check everything.

**private** **static** **void** implMontgomeryMultiplyChecks

(**int**[] a, **int**[] b, **int**[] n, **int** len, **int**[] product) **throws** RuntimeException {

**if** (len % 2 != 0) {

**throw** **new** IllegalArgumentException("input array length must be even: " + len);

}

**if** (len < 1) {

**throw** **new** IllegalArgumentException("invalid input length: " + len);

}

**if** (len > a.length ||

len > b.length ||

len > n.length ||

(product != **null** && len > product.length)) {

**throw** **new** IllegalArgumentException("input array length out of bound: " + len);

}

}

// Make sure that the int array z (which is expected to contain

// the result of a Montgomery multiplication) is present and

// sufficiently large.

**private** **static** **int**[] materialize(**int**[] z, **int** len) {

**if** (z == **null** || z.length < len)

z = **new** **int**[len];

**return** z;

}

// These methods are intended to be be replaced by virtual machine

// intrinsics.

**private** **static** **int**[] implMontgomeryMultiply(**int**[] a, **int**[] b, **int**[] n, **int** len,

**long** inv, **int**[] product) {

product = *multiplyToLen*(a, len, b, len, product);

**return** *montReduce*(product, n, len, (**int**)inv);

}

**private** **static** **int**[] implMontgomerySquare(**int**[] a, **int**[] n, **int** len,

**long** inv, **int**[] product) {

product = *squareToLen*(a, len, product);

**return** *montReduce*(product, n, len, (**int**)inv);

}

**static** **int**[] *bnExpModThreshTable* = {7, 25, 81, 241, 673, 1793,

Integer.***MAX\_VALUE***}; // Sentinel

/\*\*

\* Returns a BigInteger whose value is x to the power of y mod z.

\* Assumes: z is odd && x < z.

\*/

**private** BigInteger oddModPow(BigInteger y, BigInteger z) {

/\*

\* The algorithm is adapted from Colin Plumb's C library.

\*

\* The window algorithm:

\* The idea is to keep a running product of b1 = n^(high-order bits of exp)

\* and then keep appending exponent bits to it. The following patterns

\* apply to a 3-bit window (k = 3):

\* To append 0: square

\* To append 1: square, multiply by n^1

\* To append 10: square, multiply by n^1, square

\* To append 11: square, square, multiply by n^3

\* To append 100: square, multiply by n^1, square, square

\* To append 101: square, square, square, multiply by n^5

\* To append 110: square, square, multiply by n^3, square

\* To append 111: square, square, square, multiply by n^7

\*

\* Since each pattern involves only one multiply, the longer the pattern

\* the better, except that a 0 (no multiplies) can be appended directly.

\* We precompute a table of odd powers of n, up to 2^k, and can then

\* multiply k bits of exponent at a time. Actually, assuming random

\* exponents, there is on average one zero bit between needs to

\* multiply (1/2 of the time there's none, 1/4 of the time there's 1,

\* 1/8 of the time, there's 2, 1/32 of the time, there's 3, etc.), so

\* you have to do one multiply per k+1 bits of exponent.

\*

\* The loop walks down the exponent, squaring the result buffer as

\* it goes. There is a wbits+1 bit lookahead buffer, buf, that is

\* filled with the upcoming exponent bits. (What is read after the

\* end of the exponent is unimportant, but it is filled with zero here.)

\* When the most-significant bit of this buffer becomes set, i.e.

\* (buf & tblmask) != 0, we have to decide what pattern to multiply

\* by, and when to do it. We decide, remember to do it in future

\* after a suitable number of squarings have passed (e.g. a pattern

\* of "100" in the buffer requires that we multiply by n^1 immediately;

\* a pattern of "110" calls for multiplying by n^3 after one more

\* squaring), clear the buffer, and continue.

\*

\* When we start, there is one more optimization: the result buffer

\* is implcitly one, so squaring it or multiplying by it can be

\* optimized away. Further, if we start with a pattern like "100"

\* in the lookahead window, rather than placing n into the buffer

\* and then starting to square it, we have already computed n^2

\* to compute the odd-powers table, so we can place that into

\* the buffer and save a squaring.

\*

\* This means that if you have a k-bit window, to compute n^z,

\* where z is the high k bits of the exponent, 1/2 of the time

\* it requires no squarings. 1/4 of the time, it requires 1

\* squaring, ... 1/2^(k-1) of the time, it reqires k-2 squarings.

\* And the remaining 1/2^(k-1) of the time, the top k bits are a

\* 1 followed by k-1 0 bits, so it again only requires k-2

\* squarings, not k-1. The average of these is 1. Add that

\* to the one squaring we have to do to compute the table,

\* and you'll see that a k-bit window saves k-2 squarings

\* as well as reducing the multiplies. (It actually doesn't

\* hurt in the case k = 1, either.)

\*/

// Special case for exponent of one

**if** (y.equals(***ONE***))

**return** **this**;

// Special case for base of zero

**if** (signum == 0)

**return** ***ZERO***;

**int**[] base = mag.clone();

**int**[] exp = y.mag;

**int**[] mod = z.mag;

**int** modLen = mod.length;

// Make modLen even. It is conventional to use a cryptographic

// modulus that is 512, 768, 1024, or 2048 bits, so this code

// will not normally be executed. However, it is necessary for

// the correct functioning of the HotSpot intrinsics.

**if** ((modLen & 1) != 0) {

**int**[] x = **new** **int**[modLen + 1];

System.*arraycopy*(mod, 0, x, 1, modLen);

mod = x;

modLen++;

}

// Select an appropriate window size

**int** wbits = 0;

**int** ebits = *bitLength*(exp, exp.length);

// if exponent is 65537 (0x10001), use minimum window size

**if** ((ebits != 17) || (exp[0] != 65537)) {

**while** (ebits > *bnExpModThreshTable*[wbits]) {

wbits++;

}

}

// Calculate appropriate table size

**int** tblmask = 1 << wbits;

// Allocate table for precomputed odd powers of base in Montgomery form

**int**[][] table = **new** **int**[tblmask][];

**for** (**int** i=0; i < tblmask; i++)

table[i] = **new** **int**[modLen];

// Compute the modular inverse of the least significant 64-bit

// digit of the modulus

**long** n0 = (mod[modLen-1] & ***LONG\_MASK***) + ((mod[modLen-2] & ***LONG\_MASK***) << 32);

**long** inv = -MutableBigInteger.*inverseMod64*(n0);

// Convert base to Montgomery form

**int**[] a = *leftShift*(base, base.length, modLen << 5);

MutableBigInteger q = **new** MutableBigInteger(),

a2 = **new** MutableBigInteger(a),

b2 = **new** MutableBigInteger(mod);

b2.normalize(); // MutableBigInteger.divide() assumes that its

// divisor is in normal form.

MutableBigInteger r= a2.divide(b2, q);

table[0] = r.toIntArray();

// Pad table[0] with leading zeros so its length is at least modLen

**if** (table[0].length < modLen) {

**int** offset = modLen - table[0].length;

**int**[] t2 = **new** **int**[modLen];

System.*arraycopy*(table[0], 0, t2, offset, table[0].length);

table[0] = t2;

}

// Set b to the square of the base

**int**[] b = *montgomerySquare*(table[0], mod, modLen, inv, **null**);

// Set t to high half of b

**int**[] t = Arrays.*copyOf*(b, modLen);

// Fill in the table with odd powers of the base

**for** (**int** i=1; i < tblmask; i++) {

table[i] = *montgomeryMultiply*(t, table[i-1], mod, modLen, inv, **null**);

}

// Pre load the window that slides over the exponent

**int** bitpos = 1 << ((ebits-1) & (32-1));

**int** buf = 0;

**int** elen = exp.length;

**int** eIndex = 0;

**for** (**int** i = 0; i <= wbits; i++) {

buf = (buf << 1) | (((exp[eIndex] & bitpos) != 0)?1:0);

bitpos >>>= 1;

**if** (bitpos == 0) {

eIndex++;

bitpos = 1 << (32-1);

elen--;

}

}

**int** multpos = ebits;

// The first iteration, which is hoisted out of the main loop

ebits--;

**boolean** isone = **true**;

multpos = ebits - wbits;

**while** ((buf & 1) == 0) {

buf >>>= 1;

multpos++;

}

**int**[] mult = table[buf >>> 1];

buf = 0;

**if** (multpos == ebits)

isone = **false**;

// The main loop

**while** (**true**) {

ebits--;

// Advance the window

buf <<= 1;

**if** (elen != 0) {

buf |= ((exp[eIndex] & bitpos) != 0) ? 1 : 0;

bitpos >>>= 1;

**if** (bitpos == 0) {

eIndex++;

bitpos = 1 << (32-1);

elen--;

}

}

// Examine the window for pending multiplies

**if** ((buf & tblmask) != 0) {

multpos = ebits - wbits;

**while** ((buf & 1) == 0) {

buf >>>= 1;

multpos++;

}

mult = table[buf >>> 1];

buf = 0;

}

// Perform multiply

**if** (ebits == multpos) {

**if** (isone) {

b = mult.clone();

isone = **false**;

} **else** {

t = b;

a = *montgomeryMultiply*(t, mult, mod, modLen, inv, a);

t = a; a = b; b = t;

}

}

// Check if done

**if** (ebits == 0)

**break**;

// Square the input

**if** (!isone) {

t = b;

a = *montgomerySquare*(t, mod, modLen, inv, a);

t = a; a = b; b = t;

}

}

// Convert result out of Montgomery form and return

**int**[] t2 = **new** **int**[2\*modLen];

System.*arraycopy*(b, 0, t2, modLen, modLen);

b = *montReduce*(t2, mod, modLen, (**int**)inv);

t2 = Arrays.*copyOf*(b, modLen);

**return** **new** BigInteger(1, t2);

}

/\*\*

\* Montgomery reduce n, modulo mod. This reduces modulo mod and divides

\* by 2^(32\*mlen). Adapted from Colin Plumb's C library.

\*/

**private** **static** **int**[] montReduce(**int**[] n, **int**[] mod, **int** mlen, **int** inv) {

**int** c=0;

**int** len = mlen;

**int** offset=0;

**do** {

**int** nEnd = n[n.length-1-offset];

**int** carry = *mulAdd*(n, mod, offset, mlen, inv \* nEnd);

c += *addOne*(n, offset, mlen, carry);

offset++;

} **while** (--len > 0);

**while** (c > 0)

c += *subN*(n, mod, mlen);

**while** (*intArrayCmpToLen*(n, mod, mlen) >= 0)

*subN*(n, mod, mlen);

**return** n;

}

/\*

\* Returns -1, 0 or +1 as big-endian unsigned int array arg1 is less than,

\* equal to, or greater than arg2 up to length len.

\*/

**private** **static** **int** intArrayCmpToLen(**int**[] arg1, **int**[] arg2, **int** len) {

**for** (**int** i=0; i < len; i++) {

**long** b1 = arg1[i] & ***LONG\_MASK***;

**long** b2 = arg2[i] & ***LONG\_MASK***;

**if** (b1 < b2)

**return** -1;

**if** (b1 > b2)

**return** 1;

}

**return** 0;

}

/\*\*

\* Subtracts two numbers of same length, returning borrow.

\*/

**private** **static** **int** subN(**int**[] a, **int**[] b, **int** len) {

**long** sum = 0;

**while** (--len >= 0) {

sum = (a[len] & ***LONG\_MASK***) -

(b[len] & ***LONG\_MASK***) + (sum >> 32);

a[len] = (**int**)sum;

}

**return** (**int**)(sum >> 32);

}

/\*\*

\* Multiply an array by one word k and add to result, return the carry

\*/

**static** **int** mulAdd(**int**[] out, **int**[] in, **int** offset, **int** len, **int** k) {

*implMulAddCheck*(out, in, offset, len, k);

**return** *implMulAdd*(out, in, offset, len, k);

}

/\*\*

\* Parameters validation.

\*/

**private** **static** **void** implMulAddCheck(**int**[] out, **int**[] in, **int** offset, **int** len, **int** k) {

**if** (len > in.length) {

**throw** **new** IllegalArgumentException("input length is out of bound: " + len + " > " + in.length);

}

**if** (offset < 0) {

**throw** **new** IllegalArgumentException("input offset is invalid: " + offset);

}

**if** (offset > (out.length - 1)) {

**throw** **new** IllegalArgumentException("input offset is out of bound: " + offset + " > " + (out.length - 1));

}

**if** (len > (out.length - offset)) {

**throw** **new** IllegalArgumentException("input len is out of bound: " + len + " > " + (out.length - offset));

}

}

/\*\*

\* Java Runtime may use intrinsic for this method.

\*/

**private** **static** **int** implMulAdd(**int**[] out, **int**[] in, **int** offset, **int** len, **int** k) {

**long** kLong = k & ***LONG\_MASK***;

**long** carry = 0;

offset = out.length-offset - 1;

**for** (**int** j=len-1; j >= 0; j--) {

**long** product = (in[j] & ***LONG\_MASK***) \* kLong +

(out[offset] & ***LONG\_MASK***) + carry;

out[offset--] = (**int**)product;

carry = product >>> 32;

}

**return** (**int**)carry;

}

/\*\*

\* Add one word to the number a mlen words into a. Return the resulting

\* carry.

\*/

**static** **int** addOne(**int**[] a, **int** offset, **int** mlen, **int** carry) {

offset = a.length-1-mlen-offset;

**long** t = (a[offset] & ***LONG\_MASK***) + (carry & ***LONG\_MASK***);

a[offset] = (**int**)t;

**if** ((t >>> 32) == 0)

**return** 0;

**while** (--mlen >= 0) {

**if** (--offset < 0) { // Carry out of number

**return** 1;

} **else** {

a[offset]++;

**if** (a[offset] != 0)

**return** 0;

}

}

**return** 1;

}

/\*\*

\* Returns a BigInteger whose value is (this \*\* exponent) mod (2\*\*p)

\*/

**private** BigInteger modPow2(BigInteger exponent, **int** p) {

/\*

\* Perform exponentiation using repeated squaring trick, chopping off

\* high order bits as indicated by modulus.

\*/

BigInteger result = ***ONE***;

BigInteger baseToPow2 = **this**.mod2(p);

**int** expOffset = 0;

**int** limit = exponent.bitLength();

**if** (**this**.testBit(0))

limit = (p-1) < limit ? (p-1) : limit;

**while** (expOffset < limit) {

**if** (exponent.testBit(expOffset))

result = result.multiply(baseToPow2).mod2(p);

expOffset++;

**if** (expOffset < limit)

baseToPow2 = baseToPow2.square().mod2(p);

}

**return** result;

}

/\*\*

\* Returns a BigInteger whose value is this mod(2\*\*p).

\* Assumes that this {@code BigInteger >= 0} and {@code p > 0}.

\*/

**private** BigInteger mod2(**int** p) {

**if** (bitLength() <= p)

**return** **this**;

// Copy remaining ints of mag

**int** numInts = (p + 31) >>> 5;

**int**[] mag = **new** **int**[numInts];

System.*arraycopy*(**this**.mag, (**this**.mag.length - numInts), mag, 0, numInts);

// Mask out any excess bits

**int** excessBits = (numInts << 5) - p;

mag[0] &= (1L << (32-excessBits)) - 1;

**return** (mag[0] == 0 ? **new** BigInteger(1, mag) : **new** BigInteger(mag, 1));

}

/\*\*

\* Returns a BigInteger whose value is {@code (this}<sup>-1</sup> {@code mod m)}.

\*

\* **@param** m the modulus.

\* **@return** {@code this}<sup>-1</sup> {@code mod m}.

\* **@throws** ArithmeticException {@code m} &le; 0, or this BigInteger

\* has no multiplicative inverse mod m (that is, this BigInteger

\* is not <i>relatively prime</i> to m).

\*/

**public** BigInteger modInverse(BigInteger m) {

**if** (m.signum != 1)

**throw** **new** ArithmeticException("BigInteger: modulus not positive");

**if** (m.equals(***ONE***))

**return** ***ZERO***;

// Calculate (this mod m)

BigInteger modVal = **this**;

**if** (signum < 0 || (**this**.compareMagnitude(m) >= 0))

modVal = **this**.mod(m);

**if** (modVal.equals(***ONE***))

**return** ***ONE***;

MutableBigInteger a = **new** MutableBigInteger(modVal);

MutableBigInteger b = **new** MutableBigInteger(m);

MutableBigInteger result = a.mutableModInverse(b);

**return** result.toBigInteger(1);

}

// Shift Operations

/\*\*

\* Returns a BigInteger whose value is {@code (this << n)}.

\* The shift distance, {@code n}, may be negative, in which case

\* this method performs a right shift.

\* (Computes <tt>floor(this \* 2<sup>n</sup>)</tt>.)

\*

\* **@param** n shift distance, in bits.

\* **@return** {@code this << n}

\* **@see** #shiftRight

\*/

**public** BigInteger shiftLeft(**int** n) {

**if** (signum == 0)

**return** ***ZERO***;

**if** (n > 0) {

**return** **new** BigInteger(*shiftLeft*(mag, n), signum);

} **else** **if** (n == 0) {

**return** **this**;

} **else** {

// Possible int overflow in (-n) is not a trouble,

// because shiftRightImpl considers its argument unsigned

**return** shiftRightImpl(-n);

}

}

/\*\*

\* Returns a magnitude array whose value is {@code (mag << n)}.

\* The shift distance, {@code n}, is considered unnsigned.

\* (Computes <tt>this \* 2<sup>n</sup></tt>.)

\*

\* **@param** mag magnitude, the most-significant int ({@code mag[0]}) must be non-zero.

\* **@param** n unsigned shift distance, in bits.

\* **@return** {@code mag << n}

\*/

**private** **static** **int**[] shiftLeft(**int**[] mag, **int** n) {

**int** nInts = n >>> 5;

**int** nBits = n & 0x1f;

**int** magLen = mag.length;

**int** newMag[] = **null**;

**if** (nBits == 0) {

newMag = **new** **int**[magLen + nInts];

System.*arraycopy*(mag, 0, newMag, 0, magLen);

} **else** {

**int** i = 0;

**int** nBits2 = 32 - nBits;

**int** highBits = mag[0] >>> nBits2;

**if** (highBits != 0) {

newMag = **new** **int**[magLen + nInts + 1];

newMag[i++] = highBits;

} **else** {

newMag = **new** **int**[magLen + nInts];

}

**int** j=0;

**while** (j < magLen-1)

newMag[i++] = mag[j++] << nBits | mag[j] >>> nBits2;

newMag[i] = mag[j] << nBits;

}

**return** newMag;

}

/\*\*

\* Returns a BigInteger whose value is {@code (this >> n)}. Sign

\* extension is performed. The shift distance, {@code n}, may be

\* negative, in which case this method performs a left shift.

\* (Computes <tt>floor(this / 2<sup>n</sup>)</tt>.)

\*

\* **@param** n shift distance, in bits.

\* **@return** {@code this >> n}

\* **@see** #shiftLeft

\*/

**public** BigInteger shiftRight(**int** n) {

**if** (signum == 0)

**return** ***ZERO***;

**if** (n > 0) {

**return** shiftRightImpl(n);

} **else** **if** (n == 0) {

**return** **this**;

} **else** {

// Possible int overflow in {@code -n} is not a trouble,

// because shiftLeft considers its argument unsigned

**return** **new** BigInteger(*shiftLeft*(mag, -n), signum);

}

}

/\*\*

\* Returns a BigInteger whose value is {@code (this >> n)}. The shift

\* distance, {@code n}, is considered unsigned.

\* (Computes <tt>floor(this \* 2<sup>-n</sup>)</tt>.)

\*

\* **@param** n unsigned shift distance, in bits.

\* **@return** {@code this >> n}

\*/

**private** BigInteger shiftRightImpl(**int** n) {

**int** nInts = n >>> 5;

**int** nBits = n & 0x1f;

**int** magLen = mag.length;

**int** newMag[] = **null**;

// Special case: entire contents shifted off the end

**if** (nInts >= magLen)

**return** (signum >= 0 ? ***ZERO*** : *negConst*[1]);

**if** (nBits == 0) {

**int** newMagLen = magLen - nInts;

newMag = Arrays.*copyOf*(mag, newMagLen);

} **else** {

**int** i = 0;

**int** highBits = mag[0] >>> nBits;

**if** (highBits != 0) {

newMag = **new** **int**[magLen - nInts];

newMag[i++] = highBits;

} **else** {

newMag = **new** **int**[magLen - nInts -1];

}

**int** nBits2 = 32 - nBits;

**int** j=0;

**while** (j < magLen - nInts - 1)

newMag[i++] = (mag[j++] << nBits2) | (mag[j] >>> nBits);

}

**if** (signum < 0) {

// Find out whether any one-bits were shifted off the end.

**boolean** onesLost = **false**;

**for** (**int** i=magLen-1, j=magLen-nInts; i >= j && !onesLost; i--)

onesLost = (mag[i] != 0);

**if** (!onesLost && nBits != 0)

onesLost = (mag[magLen - nInts - 1] << (32 - nBits) != 0);

**if** (onesLost)

newMag = javaIncrement(newMag);

}

**return** **new** BigInteger(newMag, signum);

}

**int**[] javaIncrement(**int**[] val) {

**int** lastSum = 0;

**for** (**int** i=val.length-1; i >= 0 && lastSum == 0; i--)

lastSum = (val[i] += 1);

**if** (lastSum == 0) {

val = **new** **int**[val.length+1];

val[0] = 1;

}

**return** val;

}

// Bitwise Operations

/\*\*

\* Returns a BigInteger whose value is {@code (this & val)}. (This

\* method returns a negative BigInteger if and only if this and val are

\* both negative.)

\*

\* **@param** val value to be AND'ed with this BigInteger.

\* **@return** {@code this & val}

\*/

**public** BigInteger and(BigInteger val) {

**int**[] result = **new** **int**[Math.*max*(intLength(), val.intLength())];

**for** (**int** i=0; i < result.length; i++)

result[i] = (getInt(result.length-i-1)

& val.getInt(result.length-i-1));

**return** *valueOf*(result);

}

/\*\*

\* Returns a BigInteger whose value is {@code (this | val)}. (This method

\* returns a negative BigInteger if and only if either this or val is

\* negative.)

\*

\* **@param** val value to be OR'ed with this BigInteger.

\* **@return** {@code this | val}

\*/

**public** BigInteger or(BigInteger val) {

**int**[] result = **new** **int**[Math.*max*(intLength(), val.intLength())];

**for** (**int** i=0; i < result.length; i++)

result[i] = (getInt(result.length-i-1)

| val.getInt(result.length-i-1));

**return** *valueOf*(result);

}

/\*\*

\* Returns a BigInteger whose value is {@code (this ^ val)}. (This method

\* returns a negative BigInteger if and only if exactly one of this and

\* val are negative.)

\*

\* **@param** val value to be XOR'ed with this BigInteger.

\* **@return** {@code this ^ val}

\*/

**public** BigInteger xor(BigInteger val) {

**int**[] result = **new** **int**[Math.*max*(intLength(), val.intLength())];

**for** (**int** i=0; i < result.length; i++)

result[i] = (getInt(result.length-i-1)

^ val.getInt(result.length-i-1));

**return** *valueOf*(result);

}

/\*\*

\* Returns a BigInteger whose value is {@code (~this)}. (This method

\* returns a negative value if and only if this BigInteger is

\* non-negative.)

\*

\* **@return** {@code ~this}

\*/

**public** BigInteger not() {

**int**[] result = **new** **int**[intLength()];

**for** (**int** i=0; i < result.length; i++)

result[i] = ~getInt(result.length-i-1);

**return** *valueOf*(result);

}

/\*\*

\* Returns a BigInteger whose value is {@code (this & ~val)}. This

\* method, which is equivalent to {@code and(val.not())}, is provided as

\* a convenience for masking operations. (This method returns a negative

\* BigInteger if and only if {@code this} is negative and {@code val} is

\* positive.)

\*

\* **@param** val value to be complemented and AND'ed with this BigInteger.

\* **@return** {@code this & ~val}

\*/

**public** BigInteger andNot(BigInteger val) {

**int**[] result = **new** **int**[Math.*max*(intLength(), val.intLength())];

**for** (**int** i=0; i < result.length; i++)

result[i] = (getInt(result.length-i-1)

& ~val.getInt(result.length-i-1));

**return** *valueOf*(result);

}

// Single Bit Operations

/\*\*

\* Returns {@code true} if and only if the designated bit is set.

\* (Computes {@code ((this & (1<<n)) != 0)}.)

\*

\* **@param** n index of bit to test.

\* **@return** {@code true} if and only if the designated bit is set.

\* **@throws** ArithmeticException {@code n} is negative.

\*/

**public** **boolean** testBit(**int** n) {

**if** (n < 0)

**throw** **new** ArithmeticException("Negative bit address");

**return** (getInt(n >>> 5) & (1 << (n & 31))) != 0;

}

/\*\*

\* Returns a BigInteger whose value is equivalent to this BigInteger

\* with the designated bit set. (Computes {@code (this | (1<<n))}.)

\*

\* **@param** n index of bit to set.

\* **@return** {@code this | (1<<n)}

\* **@throws** ArithmeticException {@code n} is negative.

\*/

**public** BigInteger setBit(**int** n) {

**if** (n < 0)

**throw** **new** ArithmeticException("Negative bit address");

**int** intNum = n >>> 5;

**int**[] result = **new** **int**[Math.*max*(intLength(), intNum+2)];

**for** (**int** i=0; i < result.length; i++)

result[result.length-i-1] = getInt(i);

result[result.length-intNum-1] |= (1 << (n & 31));

**return** *valueOf*(result);

}

/\*\*

\* Returns a BigInteger whose value is equivalent to this BigInteger

\* with the designated bit cleared.

\* (Computes {@code (this & ~(1<<n))}.)

\*

\* **@param** n index of bit to clear.

\* **@return** {@code this & ~(1<<n)}

\* **@throws** ArithmeticException {@code n} is negative.

\*/

**public** BigInteger clearBit(**int** n) {

**if** (n < 0)

**throw** **new** ArithmeticException("Negative bit address");

**int** intNum = n >>> 5;

**int**[] result = **new** **int**[Math.*max*(intLength(), ((n + 1) >>> 5) + 1)];

**for** (**int** i=0; i < result.length; i++)

result[result.length-i-1] = getInt(i);

result[result.length-intNum-1] &= ~(1 << (n & 31));

**return** *valueOf*(result);

}

/\*\*

\* Returns a BigInteger whose value is equivalent to this BigInteger

\* with the designated bit flipped.

\* (Computes {@code (this ^ (1<<n))}.)

\*

\* **@param** n index of bit to flip.

\* **@return** {@code this ^ (1<<n)}

\* **@throws** ArithmeticException {@code n} is negative.

\*/

**public** BigInteger flipBit(**int** n) {

**if** (n < 0)

**throw** **new** ArithmeticException("Negative bit address");

**int** intNum = n >>> 5;

**int**[] result = **new** **int**[Math.*max*(intLength(), intNum+2)];

**for** (**int** i=0; i < result.length; i++)

result[result.length-i-1] = getInt(i);

result[result.length-intNum-1] ^= (1 << (n & 31));

**return** *valueOf*(result);

}

/\*\*

\* Returns the index of the rightmost (lowest-order) one bit in this

\* BigInteger (the number of zero bits to the right of the rightmost

\* one bit). Returns -1 if this BigInteger contains no one bits.

\* (Computes {@code (this == 0? -1 : log2(this & -this))}.)

\*

\* **@return** index of the rightmost one bit in this BigInteger.

\*/

**public** **int** getLowestSetBit() {

@SuppressWarnings("deprecation") **int** lsb = ~~lowestSetBit~~ - 2;

**if** (lsb == -2) { // lowestSetBit not initialized yet

lsb = 0;

**if** (signum == 0) {

lsb -= 1;

} **else** {

// Search for lowest order nonzero int

**int** i,b;

**for** (i=0; (b = getInt(i)) == 0; i++)

;

lsb += (i << 5) + Integer.*numberOfTrailingZeros*(b);

}

~~lowestSetBit~~ = lsb + 2;

}

**return** lsb;

}

// Miscellaneous Bit Operations

/\*\*

\* Returns the number of bits in the minimal two's-complement

\* representation of this BigInteger, <i>excluding</i> a sign bit.

\* For positive BigIntegers, this is equivalent to the number of bits in

\* the ordinary binary representation. (Computes

\* {@code (ceil(log2(this < 0 ? -this : this+1)))}.)

\*

\* **@return** number of bits in the minimal two's-complement

\* representation of this BigInteger, <i>excluding</i> a sign bit.

\*/

**public** **int** bitLength() {

@SuppressWarnings("deprecation") **int** n = ~~bitLength~~ - 1;

**if** (n == -1) { // bitLength not initialized yet

**int**[] m = mag;

**int** len = m.length;

**if** (len == 0) {

n = 0; // offset by one to initialize

} **else** {

// Calculate the bit length of the magnitude

**int** magBitLength = ((len - 1) << 5) + *bitLengthForInt*(mag[0]);

**if** (signum < 0) {

// Check if magnitude is a power of two

**boolean** pow2 = (Integer.*bitCount*(mag[0]) == 1);

**for** (**int** i=1; i< len && pow2; i++)

pow2 = (mag[i] == 0);

n = (pow2 ? magBitLength -1 : magBitLength);

} **else** {

n = magBitLength;

}

}

~~bitLength~~ = n + 1;

}

**return** n;

}

/\*\*

\* Returns the number of bits in the two's complement representation

\* of this BigInteger that differ from its sign bit. This method is

\* useful when implementing bit-vector style sets atop BigIntegers.

\*

\* **@return** number of bits in the two's complement representation

\* of this BigInteger that differ from its sign bit.

\*/

**public** **int** bitCount() {

@SuppressWarnings("deprecation") **int** bc = ~~bitCount~~ - 1;

**if** (bc == -1) { // bitCount not initialized yet

bc = 0; // offset by one to initialize

// Count the bits in the magnitude

**for** (**int** i=0; i < mag.length; i++)

bc += Integer.*bitCount*(mag[i]);

**if** (signum < 0) {

// Count the trailing zeros in the magnitude

**int** magTrailingZeroCount = 0, j;

**for** (j=mag.length-1; mag[j] == 0; j--)

magTrailingZeroCount += 32;

magTrailingZeroCount += Integer.*numberOfTrailingZeros*(mag[j]);

bc += magTrailingZeroCount - 1;

}

~~bitCount~~ = bc + 1;

}

**return** bc;

}

// Primality Testing

/\*\*

\* Returns {@code true} if this BigInteger is probably prime,

\* {@code false} if it's definitely composite. If

\* {@code certainty} is &le; 0, {@code true} is

\* returned.

\*

\* **@param** certainty a measure of the uncertainty that the caller is

\* willing to tolerate: if the call returns {@code true}

\* the probability that this BigInteger is prime exceeds

\* (1 - 1/2<sup>{@code certainty}</sup>). The execution time of

\* this method is proportional to the value of this parameter.

\* **@return** {@code true} if this BigInteger is probably prime,

\* {@code false} if it's definitely composite.

\*/

**public** **boolean** isProbablePrime(**int** certainty) {

**if** (certainty <= 0)

**return** **true**;

BigInteger w = **this**.abs();

**if** (w.equals(***TWO***))

**return** **true**;

**if** (!w.testBit(0) || w.equals(***ONE***))

**return** **false**;

**return** w.primeToCertainty(certainty, **null**);

}

// Comparison Operations

/\*\*

\* Compares this BigInteger with the specified BigInteger. This

\* method is provided in preference to individual methods for each

\* of the six boolean comparison operators ({@literal <}, ==,

\* {@literal >}, {@literal >=}, !=, {@literal <=}). The suggested

\* idiom for performing these comparisons is: {@code

\* (x.compareTo(y)} &lt;<i>op</i>&gt; {@code 0)}, where

\* &lt;<i>op</i>&gt; is one of the six comparison operators.

\*

\* **@param** val BigInteger to which this BigInteger is to be compared.

\* **@return** -1, 0 or 1 as this BigInteger is numerically less than, equal

\* to, or greater than {@code val}.

\*/

**public** **int** compareTo(BigInteger val) {

**if** (signum == val.signum) {

**switch** (signum) {

**case** 1:

**return** compareMagnitude(val);

**case** -1:

**return** val.compareMagnitude(**this**);

**default**:

**return** 0;

}

}

**return** signum > val.signum ? 1 : -1;

}

/\*\*

\* Compares the magnitude array of this BigInteger with the specified

\* BigInteger's. This is the version of compareTo ignoring sign.

\*

\* **@param** val BigInteger whose magnitude array to be compared.

\* **@return** -1, 0 or 1 as this magnitude array is less than, equal to or

\* greater than the magnitude aray for the specified BigInteger's.

\*/

**final** **int** compareMagnitude(BigInteger val) {

**int**[] m1 = mag;

**int** len1 = m1.length;

**int**[] m2 = val.mag;

**int** len2 = m2.length;

**if** (len1 < len2)

**return** -1;

**if** (len1 > len2)

**return** 1;

**for** (**int** i = 0; i < len1; i++) {

**int** a = m1[i];

**int** b = m2[i];

**if** (a != b)

**return** ((a & ***LONG\_MASK***) < (b & ***LONG\_MASK***)) ? -1 : 1;

}

**return** 0;

}

/\*\*

\* Version of compareMagnitude that compares magnitude with long value.

\* val can't be Long.MIN\_VALUE.

\*/

**final** **int** compareMagnitude(**long** val) {

**assert** val != Long.***MIN\_VALUE***;

**int**[] m1 = mag;

**int** len = m1.length;

**if** (len > 2) {

**return** 1;

}

**if** (val < 0) {

val = -val;

}

**int** highWord = (**int**)(val >>> 32);

**if** (highWord == 0) {

**if** (len < 1)

**return** -1;

**if** (len > 1)

**return** 1;

**int** a = m1[0];

**int** b = (**int**)val;

**if** (a != b) {

**return** ((a & ***LONG\_MASK***) < (b & ***LONG\_MASK***))? -1 : 1;

}

**return** 0;

} **else** {

**if** (len < 2)

**return** -1;

**int** a = m1[0];

**int** b = highWord;

**if** (a != b) {

**return** ((a & ***LONG\_MASK***) < (b & ***LONG\_MASK***))? -1 : 1;

}

a = m1[1];

b = (**int**)val;

**if** (a != b) {

**return** ((a & ***LONG\_MASK***) < (b & ***LONG\_MASK***))? -1 : 1;

}

**return** 0;

}

}

/\*\*

\* Compares this BigInteger with the specified Object for equality.

\*

\* **@param** x Object to which this BigInteger is to be compared.

\* **@return** {@code true} if and only if the specified Object is a

\* BigInteger whose value is numerically equal to this BigInteger.

\*/

**public** **boolean** equals(Object x) {

// This test is just an optimization, which may or may not help

**if** (x == **this**)

**return** **true**;

**if** (!(x **instanceof** BigInteger))

**return** **false**;

BigInteger xInt = (BigInteger) x;

**if** (xInt.signum != signum)

**return** **false**;

**int**[] m = mag;

**int** len = m.length;

**int**[] xm = xInt.mag;

**if** (len != xm.length)

**return** **false**;

**for** (**int** i = 0; i < len; i++)

**if** (xm[i] != m[i])

**return** **false**;

**return** **true**;

}

/\*\*

\* Returns the minimum of this BigInteger and {@code val}.

\*

\* **@param** val value with which the minimum is to be computed.

\* **@return** the BigInteger whose value is the lesser of this BigInteger and

\* {@code val}. If they are equal, either may be returned.

\*/

**public** BigInteger min(BigInteger val) {

**return** (compareTo(val) < 0 ? **this** : val);

}

/\*\*

\* Returns the maximum of this BigInteger and {@code val}.

\*

\* **@param** val value with which the maximum is to be computed.

\* **@return** the BigInteger whose value is the greater of this and

\* {@code val}. If they are equal, either may be returned.

\*/

**public** BigInteger max(BigInteger val) {

**return** (compareTo(val) > 0 ? **this** : val);

}

// Hash Function

/\*\*

\* Returns the hash code for this BigInteger.

\*

\* **@return** hash code for this BigInteger.

\*/

**public** **int** hashCode() {

**int** hashCode = 0;

**for** (**int** i=0; i < mag.length; i++)

hashCode = (**int**)(31\*hashCode + (mag[i] & ***LONG\_MASK***));

**return** hashCode \* signum;

}

/\*\*

\* Returns the String representation of this BigInteger in the

\* given radix. If the radix is outside the range from {@link

\* Character#MIN\_RADIX} to {@link Character#MAX\_RADIX} inclusive,

\* it will default to 10 (as is the case for

\* {@code Integer.toString}). The digit-to-character mapping

\* provided by {@code Character.forDigit} is used, and a minus

\* sign is prepended if appropriate. (This representation is

\* compatible with the {@link #BigInteger(String, int) (String,

\* int)} constructor.)

\*

\* **@param** radix radix of the String representation.

\* **@return** String representation of this BigInteger in the given radix.

\* **@see** Integer#toString

\* **@see** Character#forDigit

\* **@see** #BigInteger(java.lang.String, int)

\*/

**public** String toString(**int** radix) {

**if** (signum == 0)

**return** "0";

**if** (radix < Character.***MIN\_RADIX*** || radix > Character.***MAX\_RADIX***)

radix = 10;

// If it's small enough, use smallToString.

**if** (mag.length <= ***SCHOENHAGE\_BASE\_CONVERSION\_THRESHOLD***)

**return** smallToString(radix);

// Otherwise use recursive toString, which requires positive arguments.

// The results will be concatenated into this StringBuilder

StringBuilder sb = **new** StringBuilder();

**if** (signum < 0) {

*toString*(**this**.negate(), sb, radix, 0);

sb.insert(0, '-');

}

**else**

*toString*(**this**, sb, radix, 0);

**return** sb.toString();

}

/\*\* This method is used to perform toString when arguments are small. \*/

**private** String smallToString(**int** radix) {

**if** (signum == 0) {

**return** "0";

}

// Compute upper bound on number of digit groups and allocate space

**int** maxNumDigitGroups = (4\*mag.length + 6)/7;

String digitGroup[] = **new** String[maxNumDigitGroups];

// Translate number to string, a digit group at a time

BigInteger tmp = **this**.abs();

**int** numGroups = 0;

**while** (tmp.signum != 0) {

BigInteger d = *longRadix*[radix];

MutableBigInteger q = **new** MutableBigInteger(),

a = **new** MutableBigInteger(tmp.mag),

b = **new** MutableBigInteger(d.mag);

MutableBigInteger r = a.divide(b, q);

BigInteger q2 = q.toBigInteger(tmp.signum \* d.signum);

BigInteger r2 = r.toBigInteger(tmp.signum \* d.signum);

digitGroup[numGroups++] = Long.*toString*(r2.longValue(), radix);

tmp = q2;

}

// Put sign (if any) and first digit group into result buffer

StringBuilder buf = **new** StringBuilder(numGroups\**digitsPerLong*[radix]+1);

**if** (signum < 0) {

buf.append('-');

}

buf.append(digitGroup[numGroups-1]);

// Append remaining digit groups padded with leading zeros

**for** (**int** i=numGroups-2; i >= 0; i--) {

// Prepend (any) leading zeros for this digit group

**int** numLeadingZeros = *digitsPerLong*[radix]-digitGroup[i].length();

**if** (numLeadingZeros != 0) {

buf.append(*zeros*[numLeadingZeros]);

}

buf.append(digitGroup[i]);

}

**return** buf.toString();

}

/\*\*

\* Converts the specified BigInteger to a string and appends to

\* {@code sb}. This implements the recursive Schoenhage algorithm

\* for base conversions.

\* <p/>

\* See Knuth, Donald, \_The Art of Computer Programming\_, Vol. 2,

\* Answers to Exercises (4.4) Question 14.

\*

\* **@param** u The number to convert to a string.

\* **@param** sb The StringBuilder that will be appended to in place.

\* **@param** radix The base to convert to.

\* **@param** digits The minimum number of digits to pad to.

\*/

**private** **static** **void** toString(BigInteger u, StringBuilder sb, **int** radix,

**int** digits) {

/\* If we're smaller than a certain threshold, use the smallToString

method, padding with leading zeroes when necessary. \*/

**if** (u.mag.length <= ***SCHOENHAGE\_BASE\_CONVERSION\_THRESHOLD***) {

String s = u.smallToString(radix);

// Pad with internal zeros if necessary.

// Don't pad if we're at the beginning of the string.

**if** ((s.length() < digits) && (sb.length() > 0)) {

**for** (**int** i=s.length(); i < digits; i++) { // May be a faster way to

sb.append('0'); // do this?

}

}

sb.append(s);

**return**;

}

**int** b, n;

b = u.bitLength();

// Calculate a value for n in the equation radix^(2^n) = u

// and subtract 1 from that value. This is used to find the

// cache index that contains the best value to divide u.

n = (**int**) Math.*round*(Math.*log*(b \* ***LOG\_TWO*** / ***logCache***[radix]) / ***LOG\_TWO*** - 1.0);

BigInteger v = *getRadixConversionCache*(radix, n);

BigInteger[] results;

results = u.divideAndRemainder(v);

**int** expectedDigits = 1 << n;

// Now recursively build the two halves of each number.

*toString*(results[0], sb, radix, digits-expectedDigits);

*toString*(results[1], sb, radix, expectedDigits);

}

/\*\*

\* Returns the value radix^(2^exponent) from the cache.

\* If this value doesn't already exist in the cache, it is added.

\* <p/>

\* This could be changed to a more complicated caching method using

\* {@code Future}.

\*/

**private** **static** BigInteger getRadixConversionCache(**int** radix, **int** exponent) {

BigInteger[] cacheLine = *powerCache*[radix]; // volatile read

**if** (exponent < cacheLine.length) {

**return** cacheLine[exponent];

}

**int** oldLength = cacheLine.length;

cacheLine = Arrays.*copyOf*(cacheLine, exponent + 1);

**for** (**int** i = oldLength; i <= exponent; i++) {

cacheLine[i] = cacheLine[i - 1].pow(2);

}

BigInteger[][] pc = *powerCache*; // volatile read again

**if** (exponent >= pc[radix].length) {

pc = pc.clone();

pc[radix] = cacheLine;

*powerCache* = pc; // volatile write, publish

}

**return** cacheLine[exponent];

}

/\* zero[i] is a string of i consecutive zeros. \*/

**private** **static** String *zeros*[] = **new** String[64];

**static** {

*zeros*[63] =

"000000000000000000000000000000000000000000000000000000000000000";

**for** (**int** i=0; i < 63; i++)

*zeros*[i] = *zeros*[63].substring(0, i);

}

/\*\*

\* Returns the decimal String representation of this BigInteger.

\* The digit-to-character mapping provided by

\* {@code Character.forDigit} is used, and a minus sign is

\* prepended if appropriate. (This representation is compatible

\* with the {@link #BigInteger(String) (String)} constructor, and

\* allows for String concatenation with Java's + operator.)

\*

\* **@return** decimal String representation of this BigInteger.

\* **@see** Character#forDigit

\* **@see** #BigInteger(java.lang.String)

\*/

**public** String toString() {

**return** toString(10);

}

/\*\*

\* Returns a byte array containing the two's-complement

\* representation of this BigInteger. The byte array will be in

\* <i>big-endian</i> byte-order: the most significant byte is in

\* the zeroth element. The array will contain the minimum number

\* of bytes required to represent this BigInteger, including at

\* least one sign bit, which is {@code (ceil((this.bitLength() +

\* 1)/8))}. (This representation is compatible with the

\* {@link #BigInteger(byte[]) (byte[])} constructor.)

\*

\* **@return** a byte array containing the two's-complement representation of

\* this BigInteger.

\* **@see** #BigInteger(byte[])

\*/

**public** **byte**[] toByteArray() {

**int** byteLen = bitLength()/8 + 1;

**byte**[] byteArray = **new** **byte**[byteLen];

**for** (**int** i=byteLen-1, bytesCopied=4, nextInt=0, intIndex=0; i >= 0; i--) {

**if** (bytesCopied == 4) {

nextInt = getInt(intIndex++);

bytesCopied = 1;

} **else** {

nextInt >>>= 8;

bytesCopied++;

}

byteArray[i] = (**byte**)nextInt;

}

**return** byteArray;

}

/\*\*

\* Converts this BigInteger to an {@code int}. This

\* conversion is analogous to a

\* <i>narrowing primitive conversion</i> from {@code long} to

\* {@code int} as defined in section 5.1.3 of

\* <cite>The Java&trade; Language Specification</cite>:

\* if this BigInteger is too big to fit in an

\* {@code int}, only the low-order 32 bits are returned.

\* Note that this conversion can lose information about the

\* overall magnitude of the BigInteger value as well as return a

\* result with the opposite sign.

\*

\* **@return** this BigInteger converted to an {@code int}.

\* **@see** #intValueExact()

\*/

**public** **int** intValue() {

**int** result = 0;

result = getInt(0);

**return** result;

}

/\*\*

\* Converts this BigInteger to a {@code long}. This

\* conversion is analogous to a

\* <i>narrowing primitive conversion</i> from {@code long} to

\* {@code int} as defined in section 5.1.3 of

\* <cite>The Java&trade; Language Specification</cite>:

\* if this BigInteger is too big to fit in a

\* {@code long}, only the low-order 64 bits are returned.

\* Note that this conversion can lose information about the

\* overall magnitude of the BigInteger value as well as return a

\* result with the opposite sign.

\*

\* **@return** this BigInteger converted to a {@code long}.

\* **@see** #longValueExact()

\*/

**public** **long** longValue() {

**long** result = 0;

**for** (**int** i=1; i >= 0; i--)

result = (result << 32) + (getInt(i) & ***LONG\_MASK***);

**return** result;

}

/\*\*

\* Converts this BigInteger to a {@code float}. This

\* conversion is similar to the

\* <i>narrowing primitive conversion</i> from {@code double} to

\* {@code float} as defined in section 5.1.3 of

\* <cite>The Java&trade; Language Specification</cite>:

\* if this BigInteger has too great a magnitude

\* to represent as a {@code float}, it will be converted to

\* {@link Float#NEGATIVE\_INFINITY} or {@link

\* Float#POSITIVE\_INFINITY} as appropriate. Note that even when

\* the return value is finite, this conversion can lose

\* information about the precision of the BigInteger value.

\*

\* **@return** this BigInteger converted to a {@code float}.

\*/

**public** **float** floatValue() {

**if** (signum == 0) {

**return** 0.0f;

}

**int** exponent = ((mag.length - 1) << 5) + *bitLengthForInt*(mag[0]) - 1;

// exponent == floor(log2(abs(this)))

**if** (exponent < Long.***SIZE*** - 1) {

**return** longValue();

} **else** **if** (exponent > Float.***MAX\_EXPONENT***) {

**return** signum > 0 ? Float.***POSITIVE\_INFINITY*** : Float.***NEGATIVE\_INFINITY***;

}

/\*

\* We need the top SIGNIFICAND\_WIDTH bits, including the "implicit"

\* one bit. To make rounding easier, we pick out the top

\* SIGNIFICAND\_WIDTH + 1 bits, so we have one to help us round up or

\* down. twiceSignifFloor will contain the top SIGNIFICAND\_WIDTH + 1

\* bits, and signifFloor the top SIGNIFICAND\_WIDTH.

\*

\* It helps to consider the real number signif = abs(this) \*

\* 2^(SIGNIFICAND\_WIDTH - 1 - exponent).

\*/

**int** shift = exponent - FloatConsts.***SIGNIFICAND\_WIDTH***;

**int** twiceSignifFloor;

// twiceSignifFloor will be == abs().shiftRight(shift).intValue()

// We do the shift into an int directly to improve performance.

**int** nBits = shift & 0x1f;

**int** nBits2 = 32 - nBits;

**if** (nBits == 0) {

twiceSignifFloor = mag[0];

} **else** {

twiceSignifFloor = mag[0] >>> nBits;

**if** (twiceSignifFloor == 0) {

twiceSignifFloor = (mag[0] << nBits2) | (mag[1] >>> nBits);

}

}

**int** signifFloor = twiceSignifFloor >> 1;

signifFloor &= FloatConsts.***SIGNIF\_BIT\_MASK***; // remove the implied bit

/\*

\* We round up if either the fractional part of signif is strictly

\* greater than 0.5 (which is true if the 0.5 bit is set and any lower

\* bit is set), or if the fractional part of signif is >= 0.5 and

\* signifFloor is odd (which is true if both the 0.5 bit and the 1 bit

\* are set). This is equivalent to the desired HALF\_EVEN rounding.

\*/

**boolean** increment = (twiceSignifFloor & 1) != 0

&& ((signifFloor & 1) != 0 || abs().getLowestSetBit() < shift);

**int** signifRounded = increment ? signifFloor + 1 : signifFloor;

**int** bits = ((exponent + FloatConsts.***EXP\_BIAS***))

<< (FloatConsts.***SIGNIFICAND\_WIDTH*** - 1);

bits += signifRounded;

/\*

\* If signifRounded == 2^24, we'd need to set all of the significand

\* bits to zero and add 1 to the exponent. This is exactly the behavior

\* we get from just adding signifRounded to bits directly. If the

\* exponent is Float.MAX\_EXPONENT, we round up (correctly) to

\* Float.POSITIVE\_INFINITY.

\*/

bits |= signum & FloatConsts.***SIGN\_BIT\_MASK***;

**return** Float.*intBitsToFloat*(bits);

}

/\*\*

\* Converts this BigInteger to a {@code double}. This

\* conversion is similar to the

\* <i>narrowing primitive conversion</i> from {@code double} to

\* {@code float} as defined in section 5.1.3 of

\* <cite>The Java&trade; Language Specification</cite>:

\* if this BigInteger has too great a magnitude

\* to represent as a {@code double}, it will be converted to

\* {@link Double#NEGATIVE\_INFINITY} or {@link

\* Double#POSITIVE\_INFINITY} as appropriate. Note that even when

\* the return value is finite, this conversion can lose

\* information about the precision of the BigInteger value.

\*

\* **@return** this BigInteger converted to a {@code double}.

\*/

**public** **double** doubleValue() {

**if** (signum == 0) {

**return** 0.0;

}

**int** exponent = ((mag.length - 1) << 5) + *bitLengthForInt*(mag[0]) - 1;

// exponent == floor(log2(abs(this))Double)

**if** (exponent < Long.***SIZE*** - 1) {

**return** longValue();

} **else** **if** (exponent > Double.***MAX\_EXPONENT***) {

**return** signum > 0 ? Double.***POSITIVE\_INFINITY*** : Double.***NEGATIVE\_INFINITY***;

}

/\*

\* We need the top SIGNIFICAND\_WIDTH bits, including the "implicit"

\* one bit. To make rounding easier, we pick out the top

\* SIGNIFICAND\_WIDTH + 1 bits, so we have one to help us round up or

\* down. twiceSignifFloor will contain the top SIGNIFICAND\_WIDTH + 1

\* bits, and signifFloor the top SIGNIFICAND\_WIDTH.

\*

\* It helps to consider the real number signif = abs(this) \*

\* 2^(SIGNIFICAND\_WIDTH - 1 - exponent).

\*/

**int** shift = exponent - DoubleConsts.***SIGNIFICAND\_WIDTH***;

**long** twiceSignifFloor;

// twiceSignifFloor will be == abs().shiftRight(shift).longValue()

// We do the shift into a long directly to improve performance.

**int** nBits = shift & 0x1f;

**int** nBits2 = 32 - nBits;

**int** highBits;

**int** lowBits;

**if** (nBits == 0) {

highBits = mag[0];

lowBits = mag[1];

} **else** {

highBits = mag[0] >>> nBits;

lowBits = (mag[0] << nBits2) | (mag[1] >>> nBits);

**if** (highBits == 0) {

highBits = lowBits;

lowBits = (mag[1] << nBits2) | (mag[2] >>> nBits);

}

}

twiceSignifFloor = ((highBits & ***LONG\_MASK***) << 32)

| (lowBits & ***LONG\_MASK***);

**long** signifFloor = twiceSignifFloor >> 1;

signifFloor &= DoubleConsts.***SIGNIF\_BIT\_MASK***; // remove the implied bit

/\*

\* We round up if either the fractional part of signif is strictly

\* greater than 0.5 (which is true if the 0.5 bit is set and any lower

\* bit is set), or if the fractional part of signif is >= 0.5 and

\* signifFloor is odd (which is true if both the 0.5 bit and the 1 bit

\* are set). This is equivalent to the desired HALF\_EVEN rounding.

\*/

**boolean** increment = (twiceSignifFloor & 1) != 0

&& ((signifFloor & 1) != 0 || abs().getLowestSetBit() < shift);

**long** signifRounded = increment ? signifFloor + 1 : signifFloor;

**long** bits = (**long**) ((exponent + DoubleConsts.***EXP\_BIAS***))

<< (DoubleConsts.***SIGNIFICAND\_WIDTH*** - 1);

bits += signifRounded;

/\*

\* If signifRounded == 2^53, we'd need to set all of the significand

\* bits to zero and add 1 to the exponent. This is exactly the behavior

\* we get from just adding signifRounded to bits directly. If the

\* exponent is Double.MAX\_EXPONENT, we round up (correctly) to

\* Double.POSITIVE\_INFINITY.

\*/

bits |= signum & DoubleConsts.***SIGN\_BIT\_MASK***;

**return** Double.*longBitsToDouble*(bits);

}

/\*\*

\* Returns a copy of the input array stripped of any leading zero bytes.

\*/

**private** **static** **int**[] stripLeadingZeroInts(**int** val[]) {

**int** vlen = val.length;

**int** keep;

// Find first nonzero byte

**for** (keep = 0; keep < vlen && val[keep] == 0; keep++)

;

**return** java.util.Arrays.*copyOfRange*(val, keep, vlen);

}

/\*\*

\* Returns the input array stripped of any leading zero bytes.

\* Since the source is trusted the copying may be skipped.

\*/

**private** **static** **int**[] trustedStripLeadingZeroInts(**int** val[]) {

**int** vlen = val.length;

**int** keep;

// Find first nonzero byte

**for** (keep = 0; keep < vlen && val[keep] == 0; keep++)

;

**return** keep == 0 ? val : java.util.Arrays.*copyOfRange*(val, keep, vlen);

}

/\*\*

\* Returns a copy of the input array stripped of any leading zero bytes.

\*/

**private** **static** **int**[] stripLeadingZeroBytes(**byte** a[]) {

**int** byteLength = a.length;

**int** keep;

// Find first nonzero byte

**for** (keep = 0; keep < byteLength && a[keep] == 0; keep++)

;

// Allocate new array and copy relevant part of input array

**int** intLength = ((byteLength - keep) + 3) >>> 2;

**int**[] result = **new** **int**[intLength];

**int** b = byteLength - 1;

**for** (**int** i = intLength-1; i >= 0; i--) {

result[i] = a[b--] & 0xff;

**int** bytesRemaining = b - keep + 1;

**int** bytesToTransfer = Math.*min*(3, bytesRemaining);

**for** (**int** j=8; j <= (bytesToTransfer << 3); j += 8)

result[i] |= ((a[b--] & 0xff) << j);

}

**return** result;

}

/\*\*

\* Takes an array a representing a negative 2's-complement number and

\* returns the minimal (no leading zero bytes) unsigned whose value is -a.

\*/

**private** **static** **int**[] makePositive(**byte** a[]) {

**int** keep, k;

**int** byteLength = a.length;

// Find first non-sign (0xff) byte of input

**for** (keep=0; keep < byteLength && a[keep] == -1; keep++)

;

/\* Allocate output array. If all non-sign bytes are 0x00, we must

\* allocate space for one extra output byte. \*/

**for** (k=keep; k < byteLength && a[k] == 0; k++)

;

**int** extraByte = (k == byteLength) ? 1 : 0;

**int** intLength = ((byteLength - keep + extraByte) + 3) >>> 2;

**int** result[] = **new** **int**[intLength];

/\* Copy one's complement of input into output, leaving extra

\* byte (if it exists) == 0x00 \*/

**int** b = byteLength - 1;

**for** (**int** i = intLength-1; i >= 0; i--) {

result[i] = a[b--] & 0xff;

**int** numBytesToTransfer = Math.*min*(3, b-keep+1);

**if** (numBytesToTransfer < 0)

numBytesToTransfer = 0;

**for** (**int** j=8; j <= 8\*numBytesToTransfer; j += 8)

result[i] |= ((a[b--] & 0xff) << j);

// Mask indicates which bits must be complemented

**int** mask = -1 >>> (8\*(3-numBytesToTransfer));

result[i] = ~result[i] & mask;

}

// Add one to one's complement to generate two's complement

**for** (**int** i=result.length-1; i >= 0; i--) {

result[i] = (**int**)((result[i] & ***LONG\_MASK***) + 1);

**if** (result[i] != 0)

**break**;

}

**return** result;

}

/\*\*

\* Takes an array a representing a negative 2's-complement number and

\* returns the minimal (no leading zero ints) unsigned whose value is -a.

\*/

**private** **static** **int**[] makePositive(**int** a[]) {

**int** keep, j;

// Find first non-sign (0xffffffff) int of input

**for** (keep=0; keep < a.length && a[keep] == -1; keep++)

;

/\* Allocate output array. If all non-sign ints are 0x00, we must

\* allocate space for one extra output int. \*/

**for** (j=keep; j < a.length && a[j] == 0; j++)

;

**int** extraInt = (j == a.length ? 1 : 0);

**int** result[] = **new** **int**[a.length - keep + extraInt];

/\* Copy one's complement of input into output, leaving extra

\* int (if it exists) == 0x00 \*/

**for** (**int** i = keep; i < a.length; i++)

result[i - keep + extraInt] = ~a[i];

// Add one to one's complement to generate two's complement

**for** (**int** i=result.length-1; ++result[i] == 0; i--)

;

**return** result;

}

/\*

\* The following two arrays are used for fast String conversions. Both

\* are indexed by radix. The first is the number of digits of the given

\* radix that can fit in a Java long without "going negative", i.e., the

\* highest integer n such that radix\*\*n < 2\*\*63. The second is the

\* "long radix" that tears each number into "long digits", each of which

\* consists of the number of digits in the corresponding element in

\* digitsPerLong (longRadix[i] = i\*\*digitPerLong[i]). Both arrays have

\* nonsense values in their 0 and 1 elements, as radixes 0 and 1 are not

\* used.

\*/

**private** **static** **int** *digitsPerLong*[] = {0, 0,

62, 39, 31, 27, 24, 22, 20, 19, 18, 18, 17, 17, 16, 16, 15, 15, 15, 14,

14, 14, 14, 13, 13, 13, 13, 13, 13, 12, 12, 12, 12, 12, 12, 12, 12};

**private** **static** BigInteger *longRadix*[] = {**null**, **null**,

*valueOf*(0x4000000000000000L), *valueOf*(0x383d9170b85ff80bL),

*valueOf*(0x4000000000000000L), *valueOf*(0x6765c793fa10079dL),

*valueOf*(0x41c21cb8e1000000L), *valueOf*(0x3642798750226111L),

*valueOf*(0x1000000000000000L), *valueOf*(0x12bf307ae81ffd59L),

*valueOf*( 0xde0b6b3a7640000L), *valueOf*(0x4d28cb56c33fa539L),

*valueOf*(0x1eca170c00000000L), *valueOf*(0x780c7372621bd74dL),

*valueOf*(0x1e39a5057d810000L), *valueOf*(0x5b27ac993df97701L),

*valueOf*(0x1000000000000000L), *valueOf*(0x27b95e997e21d9f1L),

*valueOf*(0x5da0e1e53c5c8000L), *valueOf*( 0xb16a458ef403f19L),

*valueOf*(0x16bcc41e90000000L), *valueOf*(0x2d04b7fdd9c0ef49L),

*valueOf*(0x5658597bcaa24000L), *valueOf*( 0x6feb266931a75b7L),

*valueOf*( 0xc29e98000000000L), *valueOf*(0x14adf4b7320334b9L),

*valueOf*(0x226ed36478bfa000L), *valueOf*(0x383d9170b85ff80bL),

*valueOf*(0x5a3c23e39c000000L), *valueOf*( 0x4e900abb53e6b71L),

*valueOf*( 0x7600ec618141000L), *valueOf*( 0xaee5720ee830681L),

*valueOf*(0x1000000000000000L), *valueOf*(0x172588ad4f5f0981L),

*valueOf*(0x211e44f7d02c1000L), *valueOf*(0x2ee56725f06e5c71L),

*valueOf*(0x41c21cb8e1000000L)};

/\*

\* These two arrays are the integer analogue of above.

\*/

**private** **static** **int** *digitsPerInt*[] = {0, 0, 30, 19, 15, 13, 11,

11, 10, 9, 9, 8, 8, 8, 8, 7, 7, 7, 7, 7, 7, 7, 6, 6, 6, 6,

6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5};

**private** **static** **int** *intRadix*[] = {0, 0,

0x40000000, 0x4546b3db, 0x40000000, 0x48c27395, 0x159fd800,

0x75db9c97, 0x40000000, 0x17179149, 0x3b9aca00, 0xcc6db61,

0x19a10000, 0x309f1021, 0x57f6c100, 0xa2f1b6f, 0x10000000,

0x18754571, 0x247dbc80, 0x3547667b, 0x4c4b4000, 0x6b5a6e1d,

0x6c20a40, 0x8d2d931, 0xb640000, 0xe8d4a51, 0x1269ae40,

0x17179149, 0x1cb91000, 0x23744899, 0x2b73a840, 0x34e63b41,

0x40000000, 0x4cfa3cc1, 0x5c13d840, 0x6d91b519, 0x39aa400

};

/\*\*

\* These routines provide access to the two's complement representation

\* of BigIntegers.

\*/

/\*\*

\* Returns the length of the two's complement representation in ints,

\* including space for at least one sign bit.

\*/

**private** **int** intLength() {

**return** (bitLength() >>> 5) + 1;

}

/\* Returns sign bit \*/

**private** **int** signBit() {

**return** signum < 0 ? 1 : 0;

}

/\* Returns an int of sign bits \*/

**private** **int** signInt() {

**return** signum < 0 ? -1 : 0;

}

/\*\*

\* Returns the specified int of the little-endian two's complement

\* representation (int 0 is the least significant). The int number can

\* be arbitrarily high (values are logically preceded by infinitely many

\* sign ints).

\*/

**private** **int** getInt(**int** n) {

**if** (n < 0)

**return** 0;

**if** (n >= mag.length)

**return** signInt();

**int** magInt = mag[mag.length-n-1];

**return** (signum >= 0 ? magInt :

(n <= firstNonzeroIntNum() ? -magInt : ~magInt));

}

/\*\*

\* Returns the index of the int that contains the first nonzero int in the

\* little-endian binary representation of the magnitude (int 0 is the

\* least significant). If the magnitude is zero, return value is undefined.

\*/

**private** **int** firstNonzeroIntNum() {

**int** fn = ~~firstNonzeroIntNum~~ - 2;

**if** (fn == -2) { // firstNonzeroIntNum not initialized yet

fn = 0;

// Search for the first nonzero int

**int** i;

**int** mlen = mag.length;

**for** (i = mlen - 1; i >= 0 && mag[i] == 0; i--)

;

fn = mlen - i - 1;

~~firstNonzeroIntNum~~ = fn + 2; // offset by two to initialize

}

**return** fn;

}

/\*\* use serialVersionUID from JDK 1.1. for interoperability \*/

**private** **static** **final** **long** ***serialVersionUID*** = -8287574255936472291L;

/\*\*

\* Serializable fields for BigInteger.

\*

\* **@serialField** signum int

\* signum of this BigInteger.

\* **@serialField** magnitude int[]

\* magnitude array of this BigInteger.

\* **@serialField** bitCount int

\* number of bits in this BigInteger

\* **@serialField** bitLength int

\* the number of bits in the minimal two's-complement

\* representation of this BigInteger

\* **@serialField** lowestSetBit int

\* lowest set bit in the twos complement representation

\*/

**private** **static** **final** ObjectStreamField[] ***serialPersistentFields*** = {

**new** ObjectStreamField("signum", Integer.***TYPE***),

**new** ObjectStreamField("magnitude", **byte**[].**class**),

**new** ObjectStreamField("bitCount", Integer.***TYPE***),

**new** ObjectStreamField("bitLength", Integer.***TYPE***),

**new** ObjectStreamField("firstNonzeroByteNum", Integer.***TYPE***),

**new** ObjectStreamField("lowestSetBit", Integer.***TYPE***)

};

/\*\*

\* Reconstitute the {@code BigInteger} instance from a stream (that is,

\* deserialize it). The magnitude is read in as an array of bytes

\* for historical reasons, but it is converted to an array of ints

\* and the byte array is discarded.

\* Note:

\* The current convention is to initialize the cache fields, bitCount,

\* bitLength and lowestSetBit, to 0 rather than some other marker value.

\* Therefore, no explicit action to set these fields needs to be taken in

\* readObject because those fields already have a 0 value be default since

\* defaultReadObject is not being used.

\*/

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

/\*

\* In order to maintain compatibility with previous serialized forms,

\* the magnitude of a BigInteger is serialized as an array of bytes.

\* The magnitude field is used as a temporary store for the byte array

\* that is deserialized. The cached computation fields should be

\* transient but are serialized for compatibility reasons.

\*/

// prepare to read the alternate persistent fields

ObjectInputStream.GetField fields = s.readFields();

// Read the alternate persistent fields that we care about

**int** sign = fields.get("signum", -2);

**byte**[] magnitude = (**byte**[])fields.get("magnitude", **null**);

// Validate signum

**if** (sign < -1 || sign > 1) {

String message = "BigInteger: Invalid signum value";

**if** (fields.defaulted("signum"))

message = "BigInteger: Signum not present in stream";

**throw** **new** java.io.StreamCorruptedException(message);

}

**int**[] mag = *stripLeadingZeroBytes*(magnitude);

**if** ((mag.length == 0) != (sign == 0)) {

String message = "BigInteger: signum-magnitude mismatch";

**if** (fields.defaulted("magnitude"))

message = "BigInteger: Magnitude not present in stream";

**throw** **new** java.io.StreamCorruptedException(message);

}

// Commit final fields via Unsafe

UnsafeHolder.*putSign*(**this**, sign);

// Calculate mag field from magnitude and discard magnitude

UnsafeHolder.*putMag*(**this**, mag);

**if** (mag.length >= ***MAX\_MAG\_LENGTH***) {

**try** {

checkRange();

} **catch** (ArithmeticException e) {

**throw** **new** java.io.StreamCorruptedException("BigInteger: Out of the supported range");

}

}

}

// Support for resetting final fields while deserializing

**private** **static** **class** UnsafeHolder {

**private** **static** **final** sun.misc.Unsafe ***unsafe***;

**private** **static** **final** **long** ***signumOffset***;

**private** **static** **final** **long** ***magOffset***;

**static** {

**try** {

***unsafe*** = sun.misc.Unsafe.*getUnsafe*();

***signumOffset*** = ***unsafe***.objectFieldOffset

(BigInteger.**class**.getDeclaredField("signum"));

***magOffset*** = ***unsafe***.objectFieldOffset

(BigInteger.**class**.getDeclaredField("mag"));

} **catch** (Exception ex) {

**throw** **new** ExceptionInInitializerError(ex);

}

}

**static** **void** putSign(BigInteger bi, **int** sign) {

***unsafe***.putIntVolatile(bi, ***signumOffset***, sign);

}

**static** **void** putMag(BigInteger bi, **int**[] magnitude) {

***unsafe***.putObjectVolatile(bi, ***magOffset***, magnitude);

}

}

/\*\*

\* Save the {@code BigInteger} instance to a stream.

\* The magnitude of a BigInteger is serialized as a byte array for

\* historical reasons.

\*

\* **@serialData** two necessary fields are written as well as obsolete

\* fields for compatibility with older versions.

\*/

**private** **void** writeObject(ObjectOutputStream s) **throws** IOException {

// set the values of the Serializable fields

ObjectOutputStream.PutField fields = s.putFields();

fields.put("signum", signum);

fields.put("magnitude", magSerializedForm());

// The values written for cached fields are compatible with older

// versions, but are ignored in readObject so don't otherwise matter.

fields.put("bitCount", -1);

fields.put("bitLength", -1);

fields.put("lowestSetBit", -2);

fields.put("firstNonzeroByteNum", -2);

// save them

s.writeFields();

}

/\*\*

\* Returns the mag array as an array of bytes.

\*/

**private** **byte**[] magSerializedForm() {

**int** len = mag.length;

**int** bitLen = (len == 0 ? 0 : ((len - 1) << 5) + *bitLengthForInt*(mag[0]));

**int** byteLen = (bitLen + 7) >>> 3;

**byte**[] result = **new** **byte**[byteLen];

**for** (**int** i = byteLen - 1, bytesCopied = 4, intIndex = len - 1, nextInt = 0;

i >= 0; i--) {

**if** (bytesCopied == 4) {

nextInt = mag[intIndex--];

bytesCopied = 1;

} **else** {

nextInt >>>= 8;

bytesCopied++;

}

result[i] = (**byte**)nextInt;

}

**return** result;

}

/\*\*

\* Converts this {@code BigInteger} to a {@code long}, checking

\* for lost information. If the value of this {@code BigInteger}

\* is out of the range of the {@code long} type, then an

\* {@code ArithmeticException} is thrown.

\*

\* **@return** this {@code BigInteger} converted to a {@code long}.

\* **@throws** ArithmeticException if the value of {@code this} will

\* not exactly fit in a {@code long}.

\* **@see** BigInteger#longValue

\* **@since** 1.8

\*/

**public** **long** longValueExact() {

**if** (mag.length <= 2 && bitLength() <= 63)

**return** longValue();

**else**

**throw** **new** ArithmeticException("BigInteger out of long range");

}

/\*\*

\* Converts this {@code BigInteger} to an {@code int}, checking

\* for lost information. If the value of this {@code BigInteger}

\* is out of the range of the {@code int} type, then an

\* {@code ArithmeticException} is thrown.

\*

\* **@return** this {@code BigInteger} converted to an {@code int}.

\* **@throws** ArithmeticException if the value of {@code this} will

\* not exactly fit in a {@code int}.

\* **@see** BigInteger#intValue

\* **@since** 1.8

\*/

**public** **int** intValueExact() {

**if** (mag.length <= 1 && bitLength() <= 31)

**return** intValue();

**else**

**throw** **new** ArithmeticException("BigInteger out of int range");

}

/\*\*

\* Converts this {@code BigInteger} to a {@code short}, checking

\* for lost information. If the value of this {@code BigInteger}

\* is out of the range of the {@code short} type, then an

\* {@code ArithmeticException} is thrown.

\*

\* **@return** this {@code BigInteger} converted to a {@code short}.

\* **@throws** ArithmeticException if the value of {@code this} will

\* not exactly fit in a {@code short}.

\* **@see** BigInteger#shortValue

\* **@since** 1.8

\*/

**public** **short** shortValueExact() {

**if** (mag.length <= 1 && bitLength() <= 31) {

**int** value = intValue();

**if** (value >= Short.***MIN\_VALUE*** && value <= Short.***MAX\_VALUE***)

**return** shortValue();

}

**throw** **new** ArithmeticException("BigInteger out of short range");

}

/\*\*

\* Converts this {@code BigInteger} to a {@code byte}, checking

\* for lost information. If the value of this {@code BigInteger}

\* is out of the range of the {@code byte} type, then an

\* {@code ArithmeticException} is thrown.

\*

\* **@return** this {@code BigInteger} converted to a {@code byte}.

\* **@throws** ArithmeticException if the value of {@code this} will

\* not exactly fit in a {@code byte}.

\* **@see** BigInteger#byteValue

\* **@since** 1.8

\*/

**public** **byte** byteValueExact() {

**if** (mag.length <= 1 && bitLength() <= 31) {

**int** value = intValue();

**if** (value >= Byte.***MIN\_VALUE*** && value <= Byte.***MAX\_VALUE***)

**return** byteValue();

}

**throw** **new** ArithmeticException("BigInteger out of byte range");

}

}