BigInteger Class Reference

Pascal Levy
pascal.levy@aequans.com

February 28, 2015

1 Overview

The `BigInteger` class implements immutable arbitrary-precision integers. It provides methods to perform usual arithmetic operations, as well as modular arithmetic, GCD calculation, primality testing, prime generation, and a few other miscellaneous operations.

This class was not written with performance in mind. It rather focuses on correctness, portability, and good integration with the Cocoa Framework. If your application relies heavily on multi-precision integer computation, you will be luckier with C-oriented libraries such as GMP. But if you only need a little bit of modular arithmetic or want to occasionally generate a big prime number with a minimum development effort, then the `BigInteger` class is for you.

1.1 Installation

The `BigInteger` class comes in a header file and two universal libraries, one for each platform. To add big integer support to your project:

- Add the `BigInteger.h` file to your project.
- If you are targeting iOS, add the `libBigInteger-iOS.a` file. If you are targeting Mac OS X, add the `libBigInteger-MacOSX.a` file.
- Go to the Build Phases tab in the target settings. Open the "Link Binary With Libraries" panel and check the library file you have just added is present. If it is not, add it now by clicking +.
- Select the Build Settings tab and find the "Other Linker Flags" build setting. If it is not already present, add the flag `-ObjC` to this setting's value. This will tell the linker to link the whole `BigInteger` class into your application, even if the linker can't tell which methods are used. This is needed because Objective-C is a dynamic language and the linker can't always tell which classes and categories are used by your application code.
- Don't forget to `#import` the header file in every source file that requires the `BigInteger` class definition.

The universal library file for iOS contains five architectures: `armv7`, `armv7s` and `arm64` for device support, `i386` and `x86_64` for simulator support. The universal library file for Mac OS X contains two architectures: `i386` and `x86_64`. All functions behave the same whatever the platform, except for the hash method.

1.2 Internal representation

Big integers are represented internally as an array of digits. On 32-bit architectures, each digit occupies 16 bits; on 64-bit architectures, each digit occupies 32 bits. In both cases, digits are stored in little-endian byte order: the least significant bits are stored at the lowest addresses of the array. You can retrieve the bits of a big integer using the getBytes:length: method.

Besides the integer bits, the `BigInteger` class also stores a sign flag to indicate negative numbers. This contrasts with native types such as `int` and `long` that usually represent negative numbers using 2's complement. Due to this difference, the shiftRight: method does not behave on negative big integers as the C >> operator behaves on negative integer values. Refer to this function's documentation for more information.

1.3 Maximal value

On 32-bit architectures, a big integer is limited to 2^{31} digits of 16 bits each, yielding to a maximal value of $2^{34359738368}$. On 64-bit architectures, a big integer is limited to 2^{63} digits of 32 bits each, yielding to a maximal value of $2^{295147905179352825856}$. These are only theoretical limits though; a physical computer will run out of memory far before reaching these values.

For the sake of simplicity and performance, the `BigInteger` class does not perform any overflow control. It is up to the developer to ensure its application never involves integers larger than a few thousands of bits on a mobile platform such as an iPad or iPhone, and larger than a few millions of bits on a desktop computer.

The most sensitive function is exp:. It can produce billions of digits when the exponent grows.

1.4 Sample code

The code snippet below illustrates how using the `BigInteger` class is easy. It simply generates a 200-bit long random prime number.

```
BigInteger * p, * r;

r = [[BigInteger alloc] initWithRandomNumberOfSize:200 exact:YES];
p = [r nextProbablePrime];
NSLog(@"prime = %@", [p toRadix:10]);
[r release];
```

The first line allocates a `BigInteger` object and initializes its value with 200 random bits. The second line searches for the first prime greater than or equal to this number. The third line converts this number to its decimal representation and prints it to the debug console.

2 Adopted Protocols

2.1 NSCoding

- encodeWithCoder:
- initWithCoder:

2.2 NSCopying

- copyWithZone:

3 Functions by Task

3.1 Creating and initializing big integers

+ bigintWithBigInteger:
+ bigintWithInt32:
+ bigintWithRandomNumberOfSize:exact:
+ bigintWithString:radix:
+ bigintWithUnsignedInt32:
- initWithBigInteger:
- initWithInt32:
- initWithRandomNumberOfSize:exact:
- initWithString:radix:
- initWithUnsignedInt32:

3.2 Retrieving String Representation

- description
- toRadix:

3.3 Accessing Numeric Value

- getBytes:length:
- intValue
- longValue

3.4 Comparing Big Integers

- compare:
- hash
- isEqual:
- isEqualToBigInteger:
- isZero

3.5 Performing Arithmetic Operations

- abs
- add:
- divide:
- divide:remainder:
- exp:
- exp:modulo:
- greatestCommonDivisor:

- inverseModulo:
- isEven
- isOdd
- multiply:
- multiply:modulo:
- negate
- shiftLeft:
- shiftRight:
- sign
- sub:

3.6 Performing Bitwise Operations

- bitCount
- bitwiseNotUsingWidth:
- bitwiseAnd:
- bitwiseOr:
- bitwiseXor:

3.7 Handling Prime Numbers

- isProbablePrime
- nextProbablePrime

4 Class methods

4.1 bigintWithBigInteger:

Returns a `BigInteger` object initialized by copying the content of another given big integer.

```
+ (BigInteger *)bigintWithBigInteger:(BigInteger *)bigint
```

Parameters

bigint
: The big integer object from which to copy the content. Must not be `nil`.

Return Value

A `BigInteger` object initialized by copying the content of the *bigint* parameter.

4.2 bigintWithInt32:

Initializes and returns a big integer containing a given 32-bit signed value.

```
+ (BigInteger *)bigintWithInt32:(int32_t)x
```

Parameters

x
: The value for the new big integer.

Return Value

A `BigInteger` object containing *x*.

4.3 bigintWithRandomNumberOfSize:exact:

Initializes and returns a `BigInteger` object containing a random value.

```
+ (BigInteger *)bigintWithRandomNumberOfSize:(int)bitcount exact:(BOOL)exact
```

Parameters

bitcount
: Length in bits of the generated random number. Should be greater than or equal to 2.

exact
: Indicates whether the returned big integer should contain exactly *bitcount* bits or not. See discussion below.

Return Value

A `BigInteger` object containing a random value of the specified length.

Discussion

If the *exact* parameter is set to `YES`, the returned big integer is exactly *bitcount* bits long; in other words, its highest bit is always 1. If the *exact* parameter is set to `NO`, all the bits in the returned big integer are fully random; this implies its length may be a little shorter than *bitcount* bits if by chance the highest bits are 0's.

This method internally uses the BSD `arc4random()` pseudo-random number generator. You don't need to seed this generator as it initializes itself the first time it is called. Please refer to "Mac OS X Manual Page For ARC4RANDOM(3)" for more information.

4.4 bigintWithString:radix:

Initializes and returns a `BigInteger` object from the given string representation of an integer.

```
+ (BigInteger *)bigintWithString:(NSString *)num radix:(int)radix
```

Parameters

num
: The string representation of an integer. Must not be `nil`.

radix
: The radix to use to interpret *num*. Should lie between 2 and 36 inclusive.

Return Value

A `BigInteger` object initialized by translating the given string representation of an integer in the specified radix, or `nil` if an error occurs.

Discussion

The allowed string representation consists of an optional minus sign followed by a sequence of one or more digits in the specified radix. The string should not contain any extraneous characters, such as white spaces for example.

4.5 bigintWithUnsignedInt32:

Initializes and returns a big integer containing a given 32-bit unsigned value.

```
+ (BigInteger *)bigintWithUnsignedInt32:(uint32_t)x
```

Parameters

x
: The value for the new big integer.

Return Value

A `BigInteger` object containing *x*.

5 Instance Methods

5.1 abs

Returns the absolute value of the receiver.

```
- (BigInteger *)abs
```

Return Value

A `BigInteger` object containing the absolute value of the receiver. The returned object may be the same object as the original receiver if it already contains a positive integer.

5.2 add:

Adds the given big integer to the receiver and returns the result.

```
- (BigInteger *)add:(BigInteger *)x
```

Parameters

x
: The big integer to add to the receiver. Must not be `nil`.

Return Value

A `BigInteger` object containing the sum of the receiver and *x*.

5.3 bitCount

Returns the number of bits of the binary representation of the receiver.

```
- (int)bitCount
```

Return Value

The number of bits of the binary representation of the value the receiver contains. This can also be interpreted as the one-based index of the most significant bit.

Discussion

The `BigInteger` class does not represent negative values using 2's complement but using an extra sign bit. This sign bit is not included in the count this function returns.

5.4 bitwiseNotUsingWidth:

Returns the result of a bitwise logical NOT on the receiver.

```
- (BigInteger *)bitwiseNotUsingWidth:(int)count
```

Parameters

count

The number of bits on which the operation is performed (see discussion below). Should be greater or equal to 1.

Return Value

A `BigInteger` object containing the bitwise logical NOT of the receiver.

Discussion

Unlike native types, the `BigInteger` class does not represent numbers using a fixed number of bits. Instead, it dynamically adapts its width to drop the highest bits when they are null. This behavior may cause a problem when performing bitwise operations involving the NOT operator, because these highest bits being missing, they cannot be complemented to 1's by the NOT operator. Any subsequent operation then leads to unexpected result. For example, consider computing $x \wedge \neg y$ with $x = 10$ and $y = 2$ using a 8-bit native type and using the `BigInteger` class.

Operation	Using a fixed width native type	Using `BigInteger` objects
y	00000010	10
$\neg y$	11111101	1
x	00001010	1010
$x \wedge \neg y$	00001000	0

To avoid this problem, the `bitwiseNotUsingWidth:` method provides a *count* parameter that specifies the minimum number of bits on which the operation should be performed. If *count* is greater than the actual width of the binary representation of the receiver, leading 0's are inserted on its left to expand it up to *count* bits, prior to applying the NOT operator. If you do not care about the operation width, simply pass 1 for this parameter.

Also note that the `BigInteger` class does not represent negative values using 2's complement but using an extra sign bit. Therefore this method behaves differently than the standard ~ operator on negative numbers.

The signs of the operands are ignored. The returned value is always positive.

5.5 bitwiseAnd:

Returns the result of a bitwise logical AND between the given big integer and the receiver.

```
- (BigInteger *)bitwiseAnd:(BigInteger *)x
```

Parameters

x

The big integer to AND with the receiver. Must not be `nil`.

Return Value

A `BigInteger` object containing the bitwise logical AND of the receiver and *x*.

Discussion

The `BigInteger` class does not represent negative values using 2's complement but using an extra sign bit. Therefore this method behaves differently than the standard & operator on negative numbers.

The signs of the operands are ignored. The returned value is always positive.

5.6 bitwiseOr:

Returns the result of a bitwise logical OR between the given big integer and the receiver.

```
- (BigInteger *)bitwiseOr:(BigInteger *)x
```

Parameters

x

The big integer to OR with the receiver. Must not be `nil`.

Return Value

A `BigInteger` object containing the bitwise logical OR of the receiver and *x*.

Discussion

The `BigInteger` class does not represent negative values using 2's complement but using an extra sign bit. Therefore this method behaves differently than the standard | operator on negative numbers.

The signs of the operands are ignored. The returned value is always positive.

5.7 bitwiseXor:

Returns the result of a bitwise logical XOR between the given big integer and the receiver.

```
- (BigInteger *)bitwiseXor:(BigInteger *)x
```

Parameters

x

The big integer to XOR with the receiver. Must not be `nil`.

Return Value

A `BigInteger` object containing the bitwise logical XOR of the receiver and *x*.

Discussion

The `BigInteger` class does not represent negative values using 2's complement but using an extra sign bit. Therefore this method behaves differently than the standard ^ operator on negative numbers.

The signs of the operands are ignored. The returned value is always positive.

5.8 compare:

Returns an `NSComparisonResult` value that indicates whether the receiver is greater than, equal to, or less than a given big integer.

```
- (NSComparisonResult)compare:(BigInteger *)bigint
```

Parameters

bigint
: The big integer with which to compare the receiver. Must not be `nil`.

Return Value

`NSOrderedAscending` if the value of *bigint* is greater than the receiver's, `NSOrderedSame` if they are equal, and `NSOrderedDescending` if the value of *bigint* is less than the receiver's.

5.9 copyWithZone:

Returns a new instance that is a copy of the receiver.

```
- (id)copyWithZone:(NSZone *)zone
```

Parameters

zone
: The zone identifies an area of memory from which to allocate for the new instance. This parameter is deprecated and is present for compatibility only. You should always pass `nil`.

Return Value

A `BigInteger` object that is a copy of the receiver.

Discussion

The returned object is implicitly retained by the sender, who is therefore responsible for releasing it. You will typically never invoke this method but rather the `copy` method of `NSObject`, which will in turn call `copyWithZone:` passing `nil` for the zone.

Since the `BigInteger` class is immutable, there is no point in creating several instances holding the same value, so the actual implementation simply returns `self`. This may change in a future release.

5.10 description

Returns a string describing the content of the receiver.

```
- (NSString *)description
```

Return Value

An `NSString` object containing a textual representation of the content of the receiver. This representation consists of an optional minus sign followed by one or more groups of hexadecimal digits.

Discussion

This method is implicitly called by the Cocoa formatting functions to convert a given object into a string when they encounter the %@ format specifier. The implementation in this `BigInteger` class is rather intended for debugging and logging purposes, though. To print a big integer in a user friendly manner, prefer using the toRadix: method.

5.11 divide:

Divides the receiver by the given big integer and returns the result.

```
- (BigInteger *)divide:(BigInteger *)div
```

Parameters

div
: The divisor. Must not be `nil`.

Return Value

A `BigInteger` object containing the quotient of the receiver divided by *div*.

Discussion

The remainder of the division is lost. If you are interested in the remainder value, use the divide:remainder: method instead.

5.12 divide:remainder:

Divides the receiver by the given big integer, returns the result, and optionally returns the remainder.

```
- (BigInteger *)divide:(BigInteger *)div remainder:(BigInteger **)rem
```

Parameters

div
: The divisor. Must not be `nil`.

rem
: Upon return contains the remainder of the division. If you are not interested in the remainder, pass in `NULL` or use the divide: method.

Return Value

A `BigInteger` object containing the quotient of the receiver divided by *div*. If the *rem* parameter is not `NULL`, it is set to a `BigInteger` object containing the remainder of the division.

The sign of the remainder is always the same as the sign of the receiver.

5.13 encodeWithCoder:

Encodes the receiver using a given archiver.

```
- (void)encodeWithCoder:(NSCoder *)coder
```

Parameters

coder
: An archiver object.

Discussion

The `BigInteger` class only supports keyed archiving. If the *coder* parameter points to an encoder that does not support keyed archiving, an exception is thrown.

The encoding format is platform independent. An archive written on a 32-bit architecture can be read on a 64-bit architecture and conversely. This may prove useful when exchanging data between an iOS client application and a 64-bit server application, for example.

For more information about serializing and archiving objects, please refer to the `NSCoding` protocol in the Cocoa documentation.

5.14 exp:

Raises the receiver to the given exponent and returns the result.

```
- (BigInteger *)exp:(uint32_t)exp
```

Parameters

exp
: The exponent to which the receiver should be raised.

Return Value

A `BigInteger` object containing the value of the receiver raised to the given exponent.

Discussion

Raising an integer to an exponent can lead to huge numbers. The `BigInteger` class does not restrict the value of *exp* nor it imposes limits on the maximum length of the result; however, this method is likely to crash due to memory shortage when *exp* becomes big.

Since a negative exponent would lead to a fractional result the `BigInteger` class cannot handle, the *exp* parameter is treated as an unsigned integer. You should pay attention to implicit conversions that may take place when passing a signed integer to this parameter. For example, passing -1 will be interpreted as passing $2^{32} - 1$, which will probably lead to unexpected results.

If you need to compute a modular exponentiation (that is $a^b \bmod m$) prefer using the exp:modulo: method which performs this operation without explicitly building the intermediate result, saving both memory and processor time.

5.15 exp:modulo:

Raises the receiver to the given exponent and returns the result modulo the given modulus.

```
- (BigInteger *)exp:(BigInteger *)exp modulo:(BigInteger *)mod
```

Parameters

exp
: The exponent to which the receiver should be raised. Should not be `nil`.

mod
: The modulus. Should not be `nil`.

Return Value

A `BigInteger` object containing the value of the receiver raised to the given exponent modulo the given modulus.

Discussion

Both the exponent and the modulus are expected to be positive. If they are not, the function raises an exception. Moreover, the modulus should not be zero. The return value is always positive and normalised between 0 and (*modulus* - 1).

5.16 getBytes:length:

Copies the content of the receiver to a byte array.

```
- (void)getBytes:(uint8_t *)bytes length:(int)length
```

Parameters

bytes
: A pointer to a C byte array. Must not be `NULL`.

length
: The length of the array the *bytes* parameter points to.

Discussion

This method copies the binary representation of a big integer object into the given C byte array. Data comes in the little endian byte order; in other words, the lowest bits of the big integer value go in the lowest indexes of the array, while the highest bits of the value go in the highest indexes of the array.

If the binary representation of the big integer is shorter than length bytes, the array is filled up with zeroes. On the other hand, if it is longer, only the first *length* bytes are copied into the array and the highest bits are lost.

5.17 greatestCommonDivisor:

Determines the greatest common divisor of the receiver and the given big integer.

```
- (BigInteger *)greatestCommonDivisor:(BigInteger *)bigint
```

Parameters

bigint
: A big integer value.

Return Value

A `BigInteger` object containing the greatest common divisor of the receiver and *bigint*.

Discussion

The operation is performed using the Euclid's algorithm. Both the receiver and the *bigint* parameter are expected to be strictly positive. If one of them is negative or null, the function raises an exception.

5.18 hash

Returns an unsigned integer that can be used as a hash table address.

```
- (NSUInteger)hash
```

Return Value

An unsigned integer that can be used as a hash table address.

Discussion

Two `BigInteger` objects holding the same value (i.e. calling the compare: method on them returns `NSOrderedSame`) are guaranteed to have the same hash. The reverse is not true however; two `BigInteger` objects having the same hash may hold different values.

Since the definition of `NSUInteger` by Cocoa depends on the platform, this function does not return the same hash value on 32-bit and 64-bit architectures. You should not therefore exchange the value returned by this function between platforms having different integer sizes.

5.19 initWithBigInteger:

Returns a `BigInteger` object initialized by copying the content of another given big integer.

```
- (id)initWithBigInteger:(BigInteger *)bigint
```

Parameters

bigint
: The big integer object from which to copy the content. Must not be `nil`.

Return Value

A `BigInteger` object initialized by copying the content of the *bigint* parameter.

5.20 initWithCoder:

Returns an object initialized from data in a given unarchiver.

```
- (id)initWithCoder:(NSCoder *)decoder
```

Parameters

decoder
: An unarchiver object.

Return Value

A `BigInteger` object initialized with data in *decoder*.

Discussion

The `BigInteger` class only supports keyed archiving. If the *decoder* parameter points to an encoder that does not support keyed archiving, an exception is thrown.

The encoding format is platform independent. An archive written on a 32-bit architecture can be read on a 64-bit architecture and conversely. This may prove useful when exchanging data between an iOS client application and a 64-bit server application, for example.

For more information about serializing and archiving objects, please refer to the `NSCoding` protocol in the Cocoa documentation.

5.21 initWithInt32:

Initializes and returns a big integer containing a given 32-bit signed value.

```
- (id)initWithInt32:(int32_t)x
```

Parameters

x
: The value for the new big integer.

Return Value

A `BigInteger` object containing *x*.

5.22 initWithRandomNumberOfSize:exact:

Initializes and returns a `BigInteger` object containing a random value.

```
- (id)initWithRandomNumberOfSize:(int)bitcount exact:(BOOL)exact
```

Parameters

bitcount
: Length in bits of the generated random number. Should be greater than or equal to 2.

exact
: Indicates whether the returned big integer should contain exactly bitcount bits or not. See discussion below.

Return Value

A `BigInteger` object containing a random value of the specified length.

Discussion

If the *exact* parameter is set to `YES`, the returned big integer is exactly *bitcount* bits long; in other words, its highest bit is always 1. If the *exact* parameter is set to `NO`, all the bits in the returned big integer are fully random; this implies its length may be a little shorter than *bitcount* bits if by chance the highest bits are 0's.

This method internally uses the BSD `arc4random()` pseudo-random number generator. You don't need to seed this generator as it initializes itself the first time it is called. Please refer to "Mac OS X Manual Page For ARC4RANDOM(3)" for more information.

5.23 initWithString:radix:

Initializes and returns a `BigInteger` object from the given string representation of an integer.

```
- (id)initWithString:(NSString *)num radix:(int)radix
```

Parameters

num
: The string representation of an integer. Must not be `nil`.

radix
: The radix to use to interpret *num*. Should lie between 2 and 36 inclusive.

Return Value

A `BigInteger` object initialized by translating the given string representation of an integer in the specified radix, or `nil` if an error occurs.

Discussion

The allowed string representation consists of an optional minus sign followed by a sequence of one or more digits in the specified radix. The string should not contain any extraneous characters, such as white spaces for example.

5.24 initWithUnsignedInt32:

Initializes and returns a big integer containing a given 32-bit unsigned value.

```
- (id)initWithUnsignedInt32:(uint32_t)x
```

Parameters

x
: The value for the new big integer.

Return Value

A `BigInteger` object containing *x*.

5.25 intValue

Returns the value of the receiver as a 32-bit signed integer value.

```
- (int32_t)intValue
```

Return Value

The integer value of the receiver.

Discussion

If the receiver contains a value that does not fit into a 32-bit signed integer, an exception is raised. A 32-bit signed integer can represent values from -2^{31} to $2^{31} - 1$ inclusive.

5.26 inverseModulo:

Computes the modular multiplicative inverse of the receiver.

```
- (BigInteger *)inverseModulo:(BigInteger *)mod
```

Parameters

mod
: The modulus. Must not be `nil`.

Return Value

A `BigInteger` object containing a value such as multiplying this value by the receiver modulo *mod* yields 1, or `nil` if this value does not exist.

Discussion

The operation is performed using the extended Euclid's algorithm. The *mod* parameter is expected to be strictly positive. If it is negative or null, an exception is raised.

The modular multiplicative inverse is only defined if the receiver and *mod* are relatively primes, that is $\gcd(receiver, mod) = 1$. In case the modular inverse is undefined, this function returns `nil`.

5.27 isEqual:

Indicates whether the receiver and a given object are equal.

```
- (BOOL)isEqual:(id)object
```

Parameters

object
: The object to be compared to the receiver.

Return Value

`YES` if *object* is a `BigInteger` object containing the same integer value than the receiver; `NO` otherwise.

Discussion

Two `BigInteger` objects are equal if they contain the same integer value, that is calling the compare: function on them returns `NSOrderedSame`.

This function inherited from `NSObject` is intended to compare any kind of objects. If you wish to specifically compare two `BigInteger` objects, prefer using the faster isEqualToBigInteger: function.

5.28 isEqualToBigInteger:

Indicates whether the receiver and a given big integer are equal.

```
- (BOOL)isEqualToBigInteger:(BigInteger *)bigint
```

Parameters

bigint
: The `BigInteger` object to be compared to the receiver.

Return Value

`YES` if *bigint* is not `nil` and *bigint* and the receiver contain the same integer value; `NO` otherwise.

Discussion

Two `BigInteger` objects are equal if they contain the same integer value, that is calling the compare: function on them returns `NSOrderedSame`.

5.29 isEven

Indicates whether the receiver contains an even value.

```
- (BOOL)isEven
```

Return Value

`YES` if the receiver contains an even value, that is a value whose the lowest bit is 0; `NO` otherwise.

5.30 isOdd

Indicates whether the receiver contains an odd value.

```
- (BOOL)isOdd
```

Return Value

`YES` if the receiver contains an odd value, that is a value whose the lowest bit is 1; `NO` otherwise.

5.31 isProbablePrime

Determines whether the receiver contains a prime number, using a probabilistic primality test.

```
- (BOOL)isProbablePrime
```

Return Value

`YES` if the receiver contains an integer value that is a probable prime; `NO` otherwise.

Discussion

The receiver is first checked against a table of small primes, to immediately reject numbers that are trivial multiples of 2, 3, 5, 7, 11 and so on. Then, several Miller-Rabin trials are performed. If none succeeds, the function concludes the number is probably prime.

The number of Miller-Rabin trials depends on the length of the number to test and ranges from 5 to 30. The shorter the bit count, the greater the number of trials. This ensures the probability of a false positive stays below 2^{-80}, which should be enough for most applications.

5.32 isZero

Indicates whether the receiver contains zero or not.

```
- (BOOL)isZero
```

Return Value

`YES` if the receiver contains zero; `NO` otherwise.

Discussion

Using this method is much faster and less error-prone than extracting the content of the receiver with intValue or longValue and comparing the result with zero.

5.33 longValue

Returns the value of the receiver as a 64-bit signed integer value.

```
- (int64_t)longValue
```

Return Value

The value of the receiver.

Discussion

If the receiver contains a value that does not fit into a 64-bit signed integer, an exception is raised. A 64-bit signed integer can represent values from -2^{63} to $2^{63} - 1$ inclusive.

5.34 multiply:

Multiplies the receiver with a given big integer and returns the result.

```
- (BigInteger *)multiply:(BigInteger *)mul
```

Parameters

mul
: The big integer to multiply with the receiver. Must not be `nil`.

Return Value

The result of the multiplication of the receiver by *mul*.

5.35 multiply:modulo:

Multiplies the receiver with a given big integer modulo a given modulus.

```
- (BigInteger *)multiply:(BigInteger *)mul modulo:(BigInteger *)mod
```

Parameters

mul
: The big integer to multiply with the receiver. Must not be `nil`.

mod
: The modulus. Must not be `nil`.

Return Value

The result of the multiplication of the receiver by *mul* modulo *mod*.

Discussion

The modulus is expected to be strictly positive. The function raises an exception if it is negative or null.

The return value is always positive and normalised between 0 and (modulus - 1).

5.36 negate

Returns the opposite of the receiver.

```
- (BigInteger *)negate
```

Return Value

The opposite of the receiver.

5.37 nextProbablePrime

Determines the first prime number greater than or equal to the receiver, using a probabilistic primality test.

```
- (BigInteger *)nextProbablePrime
```

Return Value

The first prime number that is greater than or equal to the receiver in absolute value.

Discussion

The function loops, enumerating all odd numbers starting with the initial receiver absolute value, until it finds a probable prime. If the receiver contains a negative value, the return value is negative. For example, calling this function on 12 returns 13, while calling it on -12 returns -13.

This function internally calls isProbablePrime to determine whether a given number is prime or composite. Refer to the documentation of that function for more information on the primality test it implements.

5.38 shiftLeft:

Shifts the bits of the receiver to the left by the specified amount.

```
- (BigInteger *)shiftLeft:(int)count
```

Parameters

count
: The number of bits by which the receiver should be shifted.

Return Value

A `BigInteger` object containing the value of the receiver shifted left by the specified amount.

5.39 shiftRight:

Shifts the bits of the receiver to the right by the specified amount.

```
- (BigInteger *)shiftRight:(int)count
```

Parameters

count
: The number of bits by which the receiver should be shifted.

Return Value

A `BigInteger` object containing the value of the receiver shifted right by the specified amount.

Discussion

Unlike native types, the `BigInteger` class does not represent negative values using 2's complement but with an extra sign bit. Therefore, shifting right negative big integers does not produce the same result as shifting right negative `int` or `long` values.

For example, $-17 >> 1$ yields -9 when performed on an `int` value, while it yields -8 when performed on a `BigInteger` object.

5.40 sign

Returns the sign of the receiver.

```
- (int)sign
```

Return Value

-1 if the receiver contains a negative value; +1 if it contains a positive value; 0 if it contains zero.

5.41 sub:

Subtract the given big integer from the receiver and returns the result.

```
- (BigInteger *)sub:(BigInteger *)x
```

Parameters

x
: The big integer to be subtracted from the receiver. Must not be `nil`.

Return Value

A `BigInteger` object containing the result of the subtraction of x from the receiver.

5.42 toRadix:

Prints the value of the receiver to a string in the specified radix.

```
- (NSString *)toRadix:(int)radix
```

Parameters

radix
: The radix. Should lie between 2 and 36.

Return Value

The textual representation of the receiver, expressed in the specified radix. This representation consists of an optional minus sign, immediately followed by one or more digits.

Discussion

The function does not insert extraneous characters, such as thousand separators.