F.L. Ţiplea • S. Iftene
C. Hriţcu • I. Goriac
R.M. Gordân • E. Erbiceanu

April 12, 2003

Faculty of Computer Science
"Al.I.Cuza" University of Iaşi
6600 Iaşi, Romania
E-mail: cripto1@infoiasi.ro

Contents

1 Introduction

MpNT is a Multi-Precision Number Theory package, developed at the Faculty of Computer Science, "Al.I.Cuza" University of Iaşi, România. The library was started as a base for cryptographic applications, and is intended to be both efficient and portable without disregarding code structure and clarity.

The library is written in ISO C++ so it should work on any available architecture. Special optimizations apply for the Intel I-32 processors under Windows and Linux. MpNT is freely available for non-commercial use. Suggestions or critics are warmly welcomed.

2 Basic policies

2.1 Programming language

We used C++ for the main part of the library because it is the fastest high-level programming language available, offering high portability at the same time. C++ retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.), and the programs are easier to understand and maintain.

A clean and intuitive interface was built using Object-Oriented Programming. This was achieved by providing the classes that best model the mathematical concepts, hiding the actual implementation. We also used other features such as guaranteed initialization of data, implicit type conversion for user defined types, user controlled memory management, and mechanisms for overloading operators, not to mention the superior type checking, default arguments and inline substitution of functions, and the reference type provided by C++.

Assembly language was used only for the small machine-dependent kernel that is intended to increase the performance of the library, because it is closest to what the machine architecture really provides. The kernel consists of the most frequently called functions. These routines are small, carefully optimized, and can be easily rewritten without much effort for different architectures. They are also written in C++, with the stated purpose of maintaining portability. Because of similarities in the number representation, the capability of using GMP [**?**] or even CLN [**?**] kernel as an alternative may be easily added.

2.2 Conventions

In order to make the interface as clean and easy to use as possible, we tried to adopt certain coding conventions. For example, the functions always have the output arguments before the inputs. The high-level functions also accept in-place arguments (the same variable can be used both as input and output). *** More to come *** or not ***

2.3 Algorithm selection

In many cases several algorithms may be used to perform the same operation depending on the operands sizes. Of course, the ones with the best O complexity are preferred when dealing with huge numbers, but on smaller operands a simpler, highly optimized algorithm may perform much better. This is why

careful performance testing is required to find out the limits of applicability for the different used algorithms.

2.4 Header files

In order to use the MpNT library, all you have to do is to include the header file `mntp.h` and to specify the namespace, as follows:

```
#include <mntp.h>
using namespace MpNT;
```

2.5 Data types

The following data types are defined in file `config.h`:

- `Flags` - data type in which every bit can be individually used to store a logical value (0 or 1).
- `Size` - signed numerical data type used for big number sizes.
- `Digit` - unsigned numerical variable representing a digit. For best performance digits should have the size of the longest microprocessor register that may be used for arithmetical operations.
- `SignedDigit` - ??????????????????????/ the most representative bit in the digit stores the sign. (for Raluca)

Furthermore, some digit-related constants are defined:

- `BYTES_PER_DIGIT` - the size of a digit (in bytes);
- `BITS_PER_DIGIT` - the size of a digit (in bits);
- `MAX_DIGIT` - the largest representable digit ($2^{\texttt{BITS\_PER\_DIGIT}} - 1$).

3 The MpInt class

An object of the MpInt class signifies a multiprecision integer, in the amount and sign representation. It can be considered an array of binary digits (the least significant digit is indexed 0). The user has access to this representation (reading and writing) either by indicating the individual bits, or by grouping them in combinations of $2, 3, 4, \ldots,$ `BITS_PER_DIGIT` thus obtaining base $2, 4, 8, 16, \ldots, 2^{\texttt{BITS\_PER\_DIGIT}}$ representations.

3.1 Interface functions

3.2 Constructors

`MpInt ()`
Default constructor that sets the number to an initial 0 value.

`MpInt (int i)`

`MpInt (long i)`

`MpInt (unsigned int ui)`

`MpInt (unsigned long ui)`
Constructors that create numbers with the `i` (`ui`) value.

`MpInt (const char* t)`
Constructor that builds an `MpInt` from a character string. (for details, see `MpInt::assign(const char* t)`)

3.3 Sign-related functions

`int MpInt::get_sign () const`
Returns the sign of the integer: -1, 0, 1, indicating that it is negative, zero or positive, respectively.

`MpInt& MpInt::abs ()`
Sets the number to its absolute value, and returns a reference to it.

`MpInt& MpInt::neg ()`
Inverts the sign of the number and returns a reference to it.

`MpInt& MpInt::set_positive ()`
Makes the number positive and returns a reference to it.

`MpInt& MpInt::set_negative ()`
Makes the number negative and returns a reference to it.

3.4 Bit-level access functions

`Size MpInt::get_length_bin () const`
Returns the binary length of the number.

`Size MpInt::get_length_bytes () const`
Returns the number of bytes in the representation.

`Size MpInt::get_length_wdig (Size w) const`
Returns the length of the number in base $2^{\texttt{w}}$.

For example, `get_length_wdig (BITS_PER_DIGIT)` returns the number of Digits used to represent the `MpInt`. If `w` ≤ 0 or `w` $>$ `BITS_PER_DIGIT` the function throws a `eWrongWindow` exception.

`bool MpInt::get_bit (Size n) const`
Returns the $(n+1)^{th}$ bit. If the number is represented on less than $n+1$ bits, the function returns zero.

`bool MpInt::get_bit_fast (Size n) const`
Returns the $(n+1)^{th}$ bit. The function does not check if the `MpInt` is large enough therefore, for incorrect inputs ($n >$ `get_length_bin`()), the result is unpredictable.

`void MpInt::set_bit (Size n, bool b)`
Sets the $(n+1)^{th}$ bit to `b`. If the `MpInt` is not large enough and `b` is `true`, the size of the number increases, and, if necessary, a memory reallocation will take place.

`void MpInt::set_bit_fast (Size n, bool b)`
Sets the $(n+1)^{th}$ bit to `b`. No checking is performed, and if `n` is out of range ($n >$ `get_length_bin`()) the result is unpredictable.

`Digit MpInt::get_wdig (Size n, Size w) const`
Returns a `w`-bit value, in which the least significant bit is the $(\mathtt{n}+1)^{th}$ bit in the binary representation of the multiprecision integer, the next is the $(\mathtt{n}+2)^{th}$ and so on. If the input `n` is out of range ($n >$ `get_length_bin`()), the function returns 0 (conditions: $0 < \mathtt{w} \leq$ `BITS_PER_DIGIT`, otherwise a `eWrongWindow` exception is thrown).

`Digit MpInt::get_wdig_fast (Size n, Size w) const`
Returns a `w`-bit value, in which the least significant bit is the $(\mathtt{n}+1)^{th}$ bit in the binary representation of the multiprecision integer, the next is the $(\mathtt{n}+2)^{th}$ and so on. If the inputs are out of range ($n >$ `get_length_bin`(), $\mathtt{w} \leq 0$ or $\mathtt{w} >$ `BITS_PER_DIGIT`) the result is unpredictable.

`void MpInt::set_wdig (Size n, Size w, Digit d)`
Copies the least significant `w` bits of `d` in the representation of the number, starting with the $(\mathtt{n}+1)^{th}$ bit. A reallocation takes place if necessary (conditions: $0 < \mathtt{w} \leq$ `BITS_PER_DIGIT`, otherwise a `eWrongWindow` exception is thrown).

`void MpInt::set_wdig_fast (Size n, Size w, Digit d)`
Copies the least significant `w` bits of `d` in the representation of the number, starting with the $(\mathtt{n}+1)^{th}$ bit. If the inputs are out of range ($n >$ `get_length_bin`(), $\mathtt{w} \leq 0$ or $\mathtt{w} >$ `BITS_PER_DIGIT`) the result is unpredictable.

`Size MpInt::get_weight () const`
Returns the number of bits set in the `MpInt`'s representation (*The Hamming weight*).

3.5 Assignment functions

`MpInt& MpInt::assign (int i)`

`MpInt& MpInt::assign (long i)`

`MpInt& MpInt::assign (unsigned int ui)`

`MpInt& MpInt::assign (unsigned long ui)`
Set the `MpInt`'s value to `i` (`ui`).

`MpInt& MpInt::assign (const char* t)`
Sets the `MpInt` to the value represented by the string `t`. The string may have the following prefixes:

- "`-`" for negative numbers
- "`+`" or nothing positive numbers
- "`b`" or "`B`" for base 2 numbers (binary representation)
- "`0`" for octal numbers

- "0x" or "0X" for base 16 numbers
- the default base is 10

For examples see [] ***Appendix

`MpInt& MpInt::assign (const MpInt& a)`
The `MpInt` takes the value of `a`.

3.6 Comparison functions

`bool MpInt::is_zero () const`
Returns `true` if the number is null.

`bool MpInt::is_positive () const`
Returns `true` if the number is positive.

`bool MpInt::is_negative () const`
Returns `true` if the number is negative.

`bool MpInt::is_even () const`
Returns `true` if the number is even.

`bool MpInt::is_odd () const`
Returns `true` if the number is odd.

`int cmp (const MpInt &a, const MpInt &b)`
Compares the operands. The returned value is -1, 0 or 1, corresponding to $a < b$, $a = b$ `and` $a > b$, respectively.

`int cmp_abs (const MpInt &a, const MpInt &b)`
Compares the absolute values of the operands. The returned value is -1, 0 or 1, corresponding to $a < b$, $a = b$ `and` $a > b$, respectively.

3.7 Bitwise operations

`void shl (MpInt &b, const MpInt &a, Digit n)`
Shifts `a` left by `n` bits, and puts the result in `b`.

`void shr (MpInt &b, const MpInt &a, Digit n)`
Shifts `a` right by `n` bits, and puts the result in `b`.

The following functions ignore the sign of each operand, and the result is always positive (they only perform operations on the amount of the `MpInt`'s)

`void bit_not (MpInt &b, const MpInt &a)`
Inverts all the bits of `a` and puts the result in `b`.

`void bit_and (MpInt &c, const MpInt &a, const MpInt &b)`
Compares each bit of `a` to the corresponding bit of `b`. If both bits are 1, the corresponding bit of `c` is set to 1, otherwise it is set to 0.

`void bit_or (MpInt &c, const MpInt &a, const MpInt &b)`
Compares each bit of `a` to the corresponding bit of `b`. If either bit is 1, the corresponding bit of `c` is set to 1, otherwise it is set to 0.

`void bit_xor (MpInt &c, const MpInt &a, const MpInt &b)`
Compares each bit of `a` to the corresponding bit of `b`. If one bit is 0 and the other bit is 1, the corresponding bit of `c` is set to 1, otherwise it is set to 0.

3.8 Basic arithmetical functions

`void add (MpInt &c, const MpInt &a, const MpInt &b)`
Adds `a` and `b` and stores the sum in `c`.

`void sub (MpInt &c, const MpInt &a, const MpInt &b)`
Subtracts `b` from `a` and stores result in `c`.

`void sqr (MpInt &b, const MpInt &a)`
Squares `a` and stores the result in `b`.

`void mul (MpInt &c, const MpInt &a, const MpInt &b)`
Multiplies `a` and `b` and stores the product in `c`.

`void div (MpInt &q, MpInt &r, const MpInt &a, const MpInt &b)`
Divides `a` by `b` and stores the quotient in `q` and the reminder in `r`. If `b` is zero, an `eDivisionByZero` exception is thrown.

`void div_r (MpInt &r, const MpInt &a, const MpInt &b)`
Divides `a` by `b` and stores the reminder in `r`. If `b` is zero, an `eDivisionByZero` exception is thrown.

`void div_q (MpInt &q, const MpInt &a, const MpInt &b)`
Divides `a` by `b` and stores the quotient in `q`. If `b` is zero, an `eDivisionByZero` exception is thrown.

3.9 Modular exponentiation functions

`void pow_mod (MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m)`
Stores $\mathtt{a}^{\mathtt{b}}$ *mod* `m` in `c`.

```
void pows_mod (MpInt &c,
    const std::vector<MpInt> &as,
    const std::vector<MpInt> &bs,
    const MpInt &m)
```
Multiexponentiation function. Sets $\mathtt{c} = (\mathtt{a_1}^{\mathtt{b_1}} * \mathtt{a_2}^{\mathtt{b_2}} * ....\mathtt{a_n}^{\mathtt{b_n}})$ *mod* `m`.

3.10 Number theory functions

`void gcd (MpInt &g, const MpInt &a, const MpInt &b)`
Sets `g` to the greatest common divisor of `a` and `b`. The result `g` is always positive, even if one or both of the operands are negative.
If both `a` and `b` are null, an `eInfiniteResult` exception is thrown.

`void gcd_ext (MpInt &g, MpInt &alpha, MpInt &beta, const MpInt &a, const MpInt &b)`
Sets `g` to the greatest common divisor of `a` and `b`. In addition, sets `alpha`

and `beta` to coefficients satisfying: `alfa` $*$ `a` $+$ `beta` $*$ `b` $=$ `g`. The result `g` is always positive, even if one or both of the operands are negative.
If both `a` and `b` are zero, an `eInfiniteResult` exception is thrown.

`void inv_mod (MpInt &b, const MpInt &a, const MpInt &m)`
Computes the multiplicative inverse of `a` modulo `m` and stores the result in `b`.

3.11 Miscellaneous functions

`friend void swap (MpInt &a, MpInt &b)`
Exchanges the values of `a` and `b` efficiently.

`void MpInt::random (Size sz)`
Generates a signed random number having `sz Digits`.

`void MpInt::random_bits (Size bit_sz)`
Generates a signed random number on `bit_sz` bits.

`void MpInt::alloc (Size new_sz)`
Allocates `new_sz Digits` for the `MpInt` and sets its value to zero. Any former value will be lost.

`bool MpInt::realloc (Size new_sz)`
Assures that the `MpInt` has at least `new_sz Digits` of allocated memory. The value of the number does not change.

`void MpInt::free ()`
Frees the memory allocated for the `MpInt`.

3.12 Overloaded operator(s)

`void MpInt::operator= (const MpInt& a)`
The `MpInt` takes the value of `a`.

3.13 Input/Output functions

`std::istream& read_bin (std::istream &in = std::cin, Size n = 0)`

`std::istream& read_oct (std::istream &in = std::cin, Size n = 0)`

`std::istream& read_dec (std::istream &in = std::cin, Size n = 0)`

`std::istream& read_hex (std::istream &in = std::cin, Size n = 0)`
Reads an `MpInt` from a stream (the standard input is the default). If the first character is '`-`', then the number represents a negative value, otherwise a positive one.

If `n` > 0, then the function reads exactly `n` symbols (the possible '`-`' or '`+`' not included) or until a symbol that does not belong to the specified base is encountered in the representation of the number to be read (in this category are also EOF, EOL, etc.). If it is null or not specified, then the reading is performed until an incorrect symbol is encountered (this is useful especially when reading from a file).

The base can be any of: 2, 8, 10 and 16.

`std::ostream& write_bin (std::ostream &out = std::cout) const`

`std::ostream& write_oct (std::ostream &out = std::cout) const`

`std::ostream& write_dec (std::ostream &out = std::cout) const`

`std::ostream& write_hex (std::ostream &out = std::cout) const`
Write an `MpInt` in base 2 (8, 10, 16, respectively) in a stream (by default the standard output). The output will consist of the sign '`-`', if needed, followed by the digits of the representation.

`Size MpInt::read_bytes (std::istream &in, Size n = 0)`

`Size MpInt::write_bytes (std::ostream &out, Size n = 0) const`
I/O functions especially designed for cryptographic purposes, which are not compatible with the ones above. The copying takes place byte by byte, directly in the number's representation, and the first byte processed (read/write) is the least significant byte of the `MpInt`. The sign of the number is disregarded.

In the `read_bytes` case for `n` = 0 (or not specified), the reading is performed until the occurrence of EOF, and for `n` > 0 exactly `n` bytes are read unless EOF is encountered, which will stop the reading. The last uncompleted digit will be zero padded.

For `write_bytes`, with `n` = 0 (or not specified) all the bytes of the number will be written. If `n` > 0, at most `n` bytes are written (if `n` > `get_length_bytes`(), then the writing process stops - there is no zero-padding).

Both functions return the number of read/written bytes.

3.14 Representation details

MpNT uses signed-magnitude representation for its multiprecision integers - objects of the `MpInt` class.

```
namespace MpNT {
     class MpInt {
     private:
        Flags_t flags;
        Size msz;
        Size sz;
        Digit *rep;

        //...
     };
}
```

The current implementation of the class includes four private data members:

- `flags` – uses a bit for storing the sign of the number: 0 for positive or null and 1 for negative. The implemented functions make sure that the number 0 never has the sign bit set. The other bits are yet unused.

- `msz` – the number of allocated `Digit`s.
- `sz` – the number of `Digit`s that are actually used in representation. For the number 0, its value must be null.
- `rep` – a pointer to the actual number representation (an array of `Digit`s). `rep[0]` stores the least significant `Digit` of the multiprecision integer. Normally, `rep[sz-1]` is not null if `sz` is strictly positive, thus avoiding unsignificant zeroes. All the bits of a `Digit` are used in representing the number, thus making it easier to write efficient assembly code.

Even though this representation uses slightly more memory, it provides quick access to class information, and the possibility of further expansion (the `flags` field may also store other information concerning the representation of the multiprecision integer).

4 The MpMod class

work in progress

5 The Kernel

Most of the kernel functions operate on unsigned arrays of `Digit`s, such as: comparisons, bit-shifts and basic arithmetical operations. However, they are risky to use because they assume that enough memory has been allocated for the results and that certain relations between operands hold. Moreover, the result cannot be stored in one of the operands, and the functions might even destroy one of them.

The kernel can be divided in two categories of functions:

- *Digit functions*, that operate on individual `Digits`;
- *Raw functions*, that operate on arrays of `Digits`.

5.1 Digit functions

To access this group of inline functions the header file `digit.h` should be included (`mpnt.h` does that automatically).

`Digit dig_add (Digit *c, Digit a, Digit b)`
: Adds `a` and `b`, stores the resulted carry in `c` and returns the sum (the initial value of the parameter `c` is disregarded)

`Digit dig_addx (Digit *c, Digit a, Digit b)`
: Adds `a`, `b` and the value of `c` (which represents a former carry), stores the new carry in `c` and returns the sum.

`Digit dig_sub (Digit *c, Digit a, Digit b)`
: Subtracts `b` from `a`, stores the borrow in `c` and returns the result (the initial value of the parameter `c` is disregarded).

`Digit dig_subx (Digit *c, Digit a, Digit b)`
Subtracts `b` and `c` (a former borrow) from `a`, stores the new borrow in `c` and returns the result.

`Digit dig_mul (Digit *r, Digit a, Digit b)`
Computes the product `a*b`, returns the least significant digit of the result, while the most significant digit is stored in `r`.

`Digit dig_div (Digit *r, Digit a, Digit b)`
Divides `a` by `b`, stores the remainder in `r` and returns the quotient.

`Digit dig_divx (Digit *r, Digit a, Digit b)`
Divides the number obtained by joining the digits specified by `r` and `a` (`ra`) by `b`, stores the resulted remainder in `r` and returns the quotient. For the result to be accurate it is compulsory that $r < b$.

5.2 Raw functions

To access these functions the header file `raw.h` should be included (`mpnt.h` does that automatically).

Raw functions operate on arrays of `Digit`s. Their arguments are pointers to the least significant `Digit`. For syntax simplification, two more data types are defined:

- `typedef const Digit *SrcPtr;`
- `typedef Digit *DestPtr;`

If the specified conditions for each function are not met, the results are unpredictable.

`void raw_fill_n (DestPtr dest, Size n, Digit val)`
Sets `n` digits, starting with the address specified by `dest`, to the value `val` (condition: $n > 0$).

`void raw_copy_up (DestPtr dest, SrcPtr src, Size n)`
Copies `n` digits from `src` to `dest`. The operation is performed starting with `src[0]` towards `src[n-1]` (condition: $n > 0$).

`void raw_copy_down (DestPtr dest, SrcPtr src, Size n)`
Copies `n` digits from `src` to `dest`. The operation is performed starting with `src[n-1]` towards `src[0]` (condition: $n > 0$).

`int raw_cmp_n (SrcPtr a, SrcPtr b, Size n)`
Compares {`a`, `n`} with {`b`, `n`}. If they are equal it returns 0, if `a` is greater 1, otherwise -1 (condition: $n > 0$).

`Digit raw_shl_n (DestPtr a, SrcPtr b, Size n, Size m)`
Shifts {`b`, `n`} to the left by `m` bits, storing the result in {`a`, `n`} and returning the most significant `m` bits of {`b`, `n`} (conditions: $n > 0$, $0 < m <$ `BITS_PER_DIGIT`).

`Digit raw_shr_n (DestPtr a, SrcPtr b, Size n, Size m)`
Shifts {b, n} to the right by m bits, storing the result in {a, n} and returning the least significant m bits of {b, n} (conditions: $n > 0$, $0 < m <$ BITS_PER_DIGIT).

`Digit raw_dadd_n (DestPtr a, DestPtr b, Size n)`
Adds {a, n} to {b, n} and stores the result at the addresses indicated by a and b, returning the last carry. This particular function is used in the Karatsuba multiplication (condition: $n > 0$).

`Digit raw_add_1 (DestPtr a, SrcPtr b, Size n, Digit d)`
Adds d to {b, n} and stores the result in {a, n}, returning the last carry (condition: $n > 0$).

`Digit raw_add_n (DestPtr a, SrcPtr b, SrcPtr c, Size n)`
Adds {b, n} and {c, n} and stores the result in {a, n}, returning the last carry (condition: $n > 0$).

`Digit raw_add (DestPtr a, SrcPtr b, Size lb, SrcPtr c, Size lc)`
Adds {b, lb} to {c, lb} and stores the result in {a,lb}, returning the last carry (conditions: $n > 0$, $lb \geq lc > 0$).

`Digit raw_sub_1 (DestPtr a, SrcPtr b, Size n, Digit d)`
Subtracts d from {b, n} and stores the result in {a, n}, returning the last borrow (condition: $n > 0$).

`Digit raw_sub_n (DestPtr a, SrcPtr b, SrcPtr c, Size n)`
Subtracts {c, n} from {b, n} and stores the result in {a, n}, returning the last borrow (condition: $n > 0$).

`Digit raw_sub (DestPtr a, SrcPtr b, Size lb, SrcPtr c, Size lc)`
Subtracts {c, lc} from {b, lb} and stores the result in {a, lb returning the last borrow (conditions: $n > 0$, $lb \geq lc > 0$).

`Digit raw_mul_1 (DestPtr w, SrcPtr u, Size n, Digit d)`
Multiplies {u, n} by d and stores the least significant n digits of result in {w, n}, returning the most significant one (condition: $n > 0$).

`Digit raw_muladd_1 (DestPtr w, SrcPtr u, Size n, Digit d)`
Multiplies {u, n} by d and adds the least significant n digits of the result to {w, n}, returning the most significant digit of the product, plus the last carry obtained in the addition (condition: $n > 0$).

`Digit raw_mulsub_1 (DestPtr w, SrcPtr u, Size n, Digit d)`
Multiplies {u, n} by d and subtracts the least significant n digits of the result from {w, n}, returning the most significant digit of the product, minus the last borrow obtained in the subtraction (condition: $n > 0$).

`void raw_sqr_base (DestPtr b, SrcPtr a, Size n)`
Squares {a, n} and stores the result in {b, 2n}. This function is used for small-sized operands (under SQR_KARA_LIMIT) (conditions: $n > 0$, {a, n} and {b, 2n} are memory zones that do not overlap).

`void raw_sqr (DestPtr b, SrcPtr a, Size n)`
Squares {a, n} and stores the result in {b, 2n}. Function used for operands of any size (conditions: n > 0, {a, n} and {b, 2n} are memory zones that do not overlap).

`void raw_mul_base (DestPtr w, SrcPtr u, Size m, SrcPtr v, Size n)`
Multiplies {u, m} by {v, n} and stores the result in {w, m+n}. This function is used for small-sized operands (under MUL_KARA_LIMIT) (conditions: n > 0, m > 0, {w, m+n} is a memory zone that does not overlap with {u, m} or {v, n}).

`void raw_mul (DestPtr c, SrcPtr a, Size la, SrcPtr b, Size lb)`
Multiplies {a, la} by {b, lb} and stores the result in {c, la+lb} (conditions: la ≥ lb > 0, {c, la+lb} is a memory zone that does not overlap with {a, la} or {b, lb}).

`Digit raw_div_1 (DestPtr q, SrcPtr a, Size la, Digit d)`
Divides {a, la} by d and writes the quotient to {q, la} and returns the remainder (conditions: la > 0, a ≥ d or {q, la} and {a, la} are memory zones that do not overlap and a has no leading zeroes).

`Digit raw_div_r_1 (SrcPtr a, Size la, Digit d)`
Divides {a, la} by d and returns the remainder (conditions: la > 0, a ≥ d and a has no leading zeroes).

`void raw_div (DestPtr q, DestPtr r, SrcPtr a, Size la, SrcPtr b, Size lb)`
Divides {a, la} by {b, lb} and stores the quotient in {q, la-lb+1} and the remainder in {r, lb} (conditions: la > 0, lb > 0, {a, la} and {b, lb}, are the only memory zones that are allowed to overlap a and b have no leading zeroes).

`void raw_div_q (DestPtr q, SrcPtr a, Size la, SrcPtr b, Size lb)`
Divides {a, la} by {b, lb} and stores the quotient in {q, la-lb+1} (conditions: la > 0, lb > 0, {q, la-lb+1} is a memory zone that does not overlap with {a, la} or {b, lb}, a and b have no leading zeroes).

`void raw_div_r (DestPtr r, SrcPtr a, Size la, SrcPtr b, Size lb)`
Divides {a, la} by {b, lb} and stores the remainder in {r, lb} (conditions: la > 0, lb > 0, {r, lb} is a memory zone that does not overlap with {a, la} or {b, lb}, a and b have no leading zeroes).

`Size raw_gcd (DestPtr g, SrcPtr a, Size la, SrcPtr b, Size lb)`
Ralucaaaaaaaaaaa

Computes in {g, ???} the greatest common divisor of the numbers {a, la} and {b, lb} and returns ??? (conditions: la > 0, lb > 0, accepta in-place??????).

exista raw_gcd_ext () ??

6 Particular notes on the algorithm implementation

6.1 Montgomery reduction

The Montgomery reduction is used especially for exponentiation. This algorithm can be used only when the modulus is odd. The prototype of the function that performs the reduction is:

`void mon_red (MpInt &b, const MpInt &a, const MpInt &m, Digit mp)`

where

- `a` is the number to be reduced,
- `m` is the modulus,
- `mp` is $-\mathtt{m}^{-1}$ *mod* (β???????????*the radix*).

The result stored in `b` is: $\mathtt{a} * \mathtt{R}^{-1}$ *mod* `m`. For computing `mp`, the function:

`void mon_pre (Digit &mp, const MpInt &m)`

can be used, where `m` is the reduction modulus.

6.2 Modular exponentiation

For this particular operation, most of the the algorithms presented in [***BookRef***] have been implemented, that allowing a comparison of their performances. The main operations used in exponentiation are the multiplication and the modular reduction. That is why the efforts were focused on their optimization. The number of multiplications is determined by the particularities of the binary representation of the exponent. These multiplications are either squarings or different operands multiplications. Because the first category turned out to be much faster (see... ??? multiplication section???), reducing the number of multiplications of different operands and increasing the use of squarings are required.

Very important in exponentiartion is the modular reduction method. This is performed either by normal division (`div_r`), or by Montgomery reduction (`mon_red`), the latter being faster, but with a significant disadvantage: it can only be applied to an odd modulus. Each function that implements the presented algorithms comes in two versions: one using `div_r`, and the other based on `mon_red`. It is recommended to use only the functions presented in ??? Cap 4 ???, which perform tests on the operands and choose the best exponentiation method. The tests showed that the best results are exhibited by the `pow_mod_swm` algorithm for an odd `m`, and by the `pow_mod_swj` algorithm for an even `m`, in case of general exponentiation, and for a fixed base, the LimLee `pow_mod` is the most efficient. For multiexponentiation, the best results were given by using the interleaving sliding window method. The limitation imposed by all exponentiation functions is that they do not accept negative operands.

6.3 Modular exponentiations that use basic techniques

The letter `m`, suffixed to the name of the exponentiation function signifies the use of `mon_red` for the modular reduction. In this case, the function receives one more parameter that can be computed by `mon_pre`.

Unless states otherwise the significance of the operands is as follows:

- `c` - is the result,
- `a` - is the base,
- `b` - the exponent,
- `m` - the modulus.
- `w` - the dimension of the bit window
- `mp` - the precalculated value used for Montgomery reduction

```
void pow_mod_bn (MpInt &c, const MpInt &a, const MpInt &b, const MpInt
    &m)
```

```
void pow_mod_bn (MpInt &c, const MpInt &a, Digit b, const MpInt &m)
```

```
void pow_mod_bnm ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m, Digit mp)
```

```
void pow_mod_bnm ( MpInt &c, const MpInt &a, Digit b, const MpInt &m,
    Digit mp)
```

These 4 functions implement the *Left-to-Right Binary* method.

```
void pow_mod_bnrl ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m)
```

```
void pow_mod_bnrlm ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m, Digit mp)
```

Functions that implement the *Right-to-Left Binary* method.

```
void pow_mod_ba ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt
    &m, Size w)
```

```
void pow_mod_bam ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m, Size w, Digit mp)
```

Functions implementing the *Left-to-Right b-ary* method.

```
void pow_mod_barl ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m, Size w)
```

```
void pow_mod_barlm ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m, Size w, Digit mp)
```

Functions implementing the *Right-to-Left b-ary* method.

```
void pow_mod_mba ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m, Size w)
```

```
void pow_mod_mbam ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m, Size w, Digit mp)
```

Functions implementing the *Modified b-ary* method.

```
void pow_mod_aba ( MpInt &c, const MpInt &a, const MpInt &b, const
    MpInt &m, Size w, const std::vector<Digit> &d)
```

`void pow_mod_abam ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m, Size w, const std::vector<Digit> &d, Digit mp)`
Functions implementing the *Adaptive b-ary* method, where `d` is the array of digits on which the precomputations are performed.

`void pow_mod_sw ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m, Size w)`

`void pow_mod_swm ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m, Size w, Digit mp)`
Implementation of the *Sliding Window* algorithm. `w` is the maximum size of the window. To find out its value, use the `swws` function, with the prototype:
`Size swws (const MpInt &a)` which takes the exponent as parameter. `w` is chosen depending of the size of the exponent, in order to minimize the number of modular reductions for a certain exponent.

`void pow_mod_swj ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m, Size w)`

`void pow_mod_swjm ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m, Size w, Digit mp)`
Implementation of the *Sliding Window Jump* algorithm for which, in a first step only half of the precomputation is performed, by computing: $\mathtt{a}^1$, $\mathtt{a}^2$, $\mathtt{a}^4$ and afterwards $\mathtt{a}^5$, $\mathtt{a}^9$, $\mathtt{a}^{13}\ldots$ At some stages in the algorithm the other odd exponent powers are computed ($\mathtt{a}^7$, $\mathtt{a}^{11}\ldots$), if necessary.

`void pow_mod_swrl ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m, Size w)`

`void pow_mod_swrlm ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m, Size w, Digit mp)`
Functions implementing the *Right-to-Left Sliding Window* method.

`void pow_mod_swlz ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m)`

`void pow_mod_swlzm ( MpInt &c, const MpInt &a, const MpInt &b, const MpInt &m, Digit mp)`
Implementation of the *Sliding Window Method Based On Data Compression*. The tree construction is the one suggested by Bocharova and Kudryashov.

6.4 Multi-exponentiation

`void pows_mod_bn (MpInt &c, const std::vector<MpInt> &as, const std::vector<MpInt> &bs, const MpInt &m)`

`void pows_mod_bnm (MpInt &c, const std::vector<MpInt> &as, const std::vector<MpInt> &bs, const MpInt &m, Digit mp)`
Implement the *Simultaneous Binary* method.

`void pows_mod_sw (MpInt &c, const std::vector<MpInt> &as, const std::vector<MpInt> &bs, const MpInt &m, Size w)`

`void pows_mod_swm (MpInt &c, const std::vector<MpInt> &as, const std::vector<MpInt> &bs, const MpInt &m, Size w, Digit mp)`
Implement the *Simultaneous Sliding Window* method.

`void pows_mod_isw (MpInt &c, const std::vector<MpInt> &as, const std::vector<MpInt> &bs, const MpInt &m)`

`void pows_mod_iswm (MpInt &c, const std::vector<MpInt> &as, const std::vector<MpInt> &bs, const MpInt &m, Digit mp)`
Implement *Interleaving Sliding Window* method.

6.5 Fixed-base exponentiation techniques

Three algorithms have been implemented, the ones put forward by Brickell-Gordon-McCurley-Wilson (BGMW), DeRooij and Lim-Lee. For each of these algorithms there are separate classes, defined in the header files: `bgmw.h`, `derooij.h` and `limlee.h`. The functions that receive `mp` as a last argument use the Montgomery method for modular reduction.

`BGMW::BGMW (const MpInt &_x, const MpInt &_m, Size _ebl, Size _w)`

`BGMW::BGMW (const MpInt &_x, const MpInt &_m, Size _ebl, Size _w, Digit mp)`

`void BGMW::recalculate (const MpInt &_x, const MpInt &_m, Size _ebl, Size _w)`

`void BGMW::recalculate (const MpInt &_x, const MpInt &_m, Size _ebl, Size _w, Digit mp)`
where:

- `_x` represents the base;
- `_m` the modulus;
- `_ebl` the binary length of the exponent;
- `_w` the size of a window (in bits).

`void BGMW::pow_mod (MpInt &xnm, const MpInt &n)`

`void BGMW::pow_mod (MpInt &xnm, const MpInt &n, Digit mp)`
Performs the actual exponentiation of base $\_x^n$, storing the result in `xnm`.

`Size BGMW::count_storage () const`
Returns the amount of memory ocupated by the precomputation (in bytes).

`DeRooij::DeRooij (const MpInt &_x, const MpInt &_m, Size _ebl)`

`void DeRooij::recalculate (const MpInt &_x, const MpInt &_m, Size _ebl)`
where:

- `_x` represents the base;
- `_m` the modulus;
- `_ebl` the binary length of the exponent.

`void DeRooij::pow_mod (MpInt &xnm, const MpInt &n)`

`void DeRooij::pow_mod (MpInt &xnm, const MpInt &n, Digit mp)`
Performs the actual exponentiation of base `_x`$^{\texttt{n}}$, storing the result in `xnm`.

`Size DeRooij::count_storage () const`
Returns the amount of memory ocupated by the precomputation (in bytes).

`LimLee::LimLee (const MpInt &_x, const MpInt &_m, Size _ebl, Size _h, Size _v)`

`LimLee::LimLee (const MpInt &_x, const MpInt &_m, Size _ebl, Size _h, Size _v, Digit mp)`

`void LimLee::recalculate (const MpInt &_x, const MpInt &_m, Size _ebl, Size _h, Size _v)`

`void LimLee::recalculate (const MpInt &_x, const MpInt &_m, Size _ebl, Size _h, Size _v, Digit mp)`
where:

- `_x` represents the base;
- `_m` the modulus;
- `_ebl` the binary length of the exponent;
- `_h` the number of large blocks;
- `_v` the number of small blocks.

`void LimLee::pow_mod (MpInt &xnm, const MpInt &n)`

`void LimLee::pow_mod (MpInt &xnm, const MpInt &n, Digit mp)`
Performs the actual exponentiation of base `_x`$^{\texttt{n}}$, storing the result in `xnm`.

`Size LimLee::count_storage () const`
Returns the amount of memory ocupated by the precomputation (in bytes).

6.6 Memory allocation

The memory management policy adopted in MpNT is based on explicit allocation of memory. To avoid frequent reallocation, when the exact amount of necessary memory is known, the user may make such a request. For the same reason, whenever reallocation occurs, we provide more space than needed. Memory may be released either on demand or automatically by the class destructors. Currently, the C++ operators `new` and `delete` are used, but we intend to allow the user to choose the allocation functions he prefers.

`DestPtr raw_alloc (Size sz)`
** work in progress ***

`DestPtr raw_realloc (DestPtr p, Size sz, Size newsz)`
** work in progress ***

`void raw_free ()`
** work in progress ***

6.7 Error handling

The following exceptions are defined and should be handled by the user:

- `eAllocationFailed` - a request for memory allocation could not be satisfied;
- `eIOError` - input/output error;
- `eDivisionByZero` - occurs during `div`, `div_r` or `div_q`;
- `eInfiniteResult` - thrown by `gcd` when both operands are null;
- `eNoMultiplicativeInverse` - occurs in `inv_mod`;
- `eWrongModulus` - occurs in the `mon_pre` function when the transmitted modulus is even;
- `eWrongExpOps` - occurs in the exponentiation functions, for operands that do not meet the specific conditions;
- `eWrongWindow` - occurs in some "`wdig`" functions, when the window size is not fit (`w` ≤ 0 or `w` $>$ `BITS_PER_DIGIT`).

7 Appendix 1

7.1 A basic program

```
// ex1.cpp
   #include <mpnt.h>
   #include <iostream>
   using namespace MpNT;
   using namespace std;

   int main ()
   {
      MpInt a, b, c;
      a = 100;
      b = 25;
      mul (c, a, b);
      c.write_dec ();
      cout << endl;
      return 0;
   }

compiled with:
g++ ex1.cpp -o ex1 -lmpnt ./ex1
will have the output:
2500
```

7.2 Assignments

```
void foo () {

   MpInt a;

   // integer assignments
   a = 0; // the instructions have the same semantic
   a.assign (0); // (a <- 0)
   a = -0x100; // a <- -256

   // character string assignments
   a = "0"; // a <- 0
   a = "b1111"; // a <- 15
   a = "0xf"; // a <- 15
   a = "017"; // a <- 15
   a = "+12345678901234567890"; // a <- 12345678901234567890
   a = "-0x0123456789abcdef"; // a <- -81985529216486895

   // pseudo-errors
   a = "b123"; // a <- 1
   a = "0x-abc"; // a <- 0
   a = "-12345abcd"; // a <- -12345
}
```

7.3 A fixed-base exponentiation example

This function calculates 10 consecutive powers of a randomly generated number `a` *mod* `m` using the LimLee algorithm and verifies the results by calling a general exponentiation function.

```
void fixed_base_exponentiation ()
{
   MpInt a, b, c, m;
   a.random_bits (64); a.abs (); // the fixed basis
   b.random_bits (64); b.abs (); // the varying exponent
   m.random_bits (64); m.abs (); // the modulus

   // the necessary precomputation for exponentiation
   LimLee ll (a, m, b.get_length_bin (), 5, 2);

   // display the used memory
   std::cout << "Bytes used " << ll.count_storage () << std::endl;

   int i;
   for (i = 0; i < 10; ++i) {
      // c ← a^b mod m
      ll.pow_mod (c, b); // the LimLee exponentiation function call
      c.write_hex ();
      std::cout << std::endl;
      pow_mod (c, a, b, m); // the general exponentiation function call
```

```
      c.write_hex ();
      std::cout << std::endl;
      sub (b, b, 1);
      if (i == 4) {
         // modify the precomputation matrix
         // (just to show that this is possible)
         ll.recalculate (a, m, b.get_length_bin (), 5, 3);
         std::cout << "Bytes used " << ll.count_storage () << std::endl;
      }
   }
}
```

Raw????? input and output (`read_bytes`, `write_bytes`)