Chapter 1

Using BigNum library

The following is the structure of an arbitrary precision integer in the BigNum library.

```
/**
 * Structure that represents big integers, d[0] is the least significant
 * digit while d[size-1] is the most significant digit
 */
typedef struct {
    bn_word maxSize; /*Maximum number of digits that d can hold*/
    bn_word size; /*number of digits*/
    char sign; /*zero->negative, other->positive*/
    bn_word *d; /*value of each digits*/
} BigInt;
```

This structure represents a common positional number system, so the Big Integer N is determined by the following expression:

$$N = \sum_{i=0}^{size-1} d_i b^i \qquad (0 \leq d_i < b)$$

where b represents the number system base, d_i are the digits which represent the number N and $size$ is the amount of digits that represent N in base b.

In BigNum library the base b is 2^B, where B is the bit length of an `int` data type in the environment used for running the library, normally this number is 16 for microcontrollers and 32 for general purpose CPUs and high level embedded systems.

Notice that the sign representation used in "BigNum" is the sign-magnitude representation, for this reason the big integer structure has a "sign" field.

It is important to mention that the $maxSize$ field establish a bound about the biggest and smallest number which can be represented, for example if a big integer T has a $maxSize$ value of 50, this big integer can hold a number k such

that $-2^{50B} < k < 2^{50B}$, pay attention to this for avoiding overflow errors in BigNum library.

1.1 BigInteger fundamentals

1.1.1 How to create an arbitrary precision number in BigNum Library?

For creating a new big integer with this library, you can use the following function:

```
BigInt* bnNewBigInt(unsigned int maxSize, unsigned int initVal);
```

Purpose :

To create a Big Integer with dynamic memory allocation.

Arguments :

- **maxSize:** Is the biggest amount of base 2^B digits (where B is the bit length of an `int` data type) that this big integer can handle, remember that this number represents a constraint about the biggest and smallest number that can be represented in the big integer,therefore it must be chosen with care because you can not change it after the big integer creation.

- **initVal:** Is the initial value of the new big integer, this number must satisfy the following: $0 \leq initval < 2^B$, the reason of this is because the data type of this argument is an `unsigned int`, for initializing the big integer with numbers bigger than 2^B or with negative numbers review the function `bnStrToInt`.

Return Value :

Returns a pointer to the new big integer created by this function.

Example :

```
BigInt *a=bnNewBigInt(50, 0);
```

this example builds a big integer with 50 base b digits with an initial value of 0.

1.1.2 How to initialize an arbitrary precision number in BigNum Library?

In the last section it was told that the big integer creation function has a constrained mechanism for initializing the big integer, in general it is possible to use the following function for this purpose.

```
void bnStrToInt(BigInt *ans, const char *input)
```

Purpose: To get a *big integer* representation of a number represented as a characters array.

Arguments:

- **ans:** it is a pointer to the `BigInt` struct which will save the number represented by the characters array `input`, the big integer pointed by `ans` must have enough size in digits, if this is not the case the conversion can not be achieved.
- **input:** it is a pointer to a characters array, this array represents a number in the common way (e.g. "-1234"), it is is important that the end of the number is represented by the *end of line* character (`'\0'`).

Return Value :

None.

Examples:

- ```
  BigInt *a=bnNewBigInt(50, 0);
  bnStrToInt(a, "-123456789000");
  ```
- ```
  BigInt *a=bnNewBigInt(50, 0);
  char *input="-123456789000";
  bnStrToInt(a, input);
  ```

  In both cases the big integer `'a'` will represent the number `-123456789000`.

1.1.3 How to get a conventional and readable representation of a BigNum arbitrary precision number?

It is useful to get an appropriate representation of a big integer for the human beings after all the operations required have been applied to the number, taking into account that the *BigNum* number representation is not very natural because of some aspects as the numerical base, the following function gives a

mechanism for getting a decimal representation of a big integer.

```
void bnIntToStr(char* ans, BigInt* x)
```

Purpose: To get a decimal representation saved in a character array from a big integer.

Arguments:

- **ans:**

  Pointer to the characters array where the decimal representation of the big integer `'x'` will be saved, keep in mind that the array size must be enough for saving the decimal representation of `'x'`. This function appends an *end of line* character to the array pointed by ans.

- **x:**

  Pointer to the big integer whose decimal representation is desired.

Return Value :

None.

Examples:

- ```
  char ans[100];
  BigInt *a=bnNewBigInt(50, 0);
  char *input="-123456789000";
  bnStrToInt(a, input);
  bnIntToStr(ans,a);
  ```

After executing this code the value of the array pointed by `ans` will be `"-123456789000"`.

1.1.4 How to free the memory assigned to a big integer?

It is very important to remember that you must free the allocated memory for a big integer when it is not needed anymore, because the memory allocation is dynamic, therefore you must explicitly free that memory space for using it in the future.

```
void bnIntToStr(char* ans, BigInt* x)
```

Purpose: To get a decimal representation saved in a character array from a big integer.
```

Arguments:

- **ans:**

  Pointer to the characters array where the decimal representation of the big integer `'x'` will be saved, keep in mind that the array size must be enough for saving the decimal representation of `'x'`. This function appends an *end of line* character to the array pointed by ans.

- **x:**

  Pointer to the big integer whose decimal representation is desired.

Return Value :

None.

Examples:

- ```
  char ans[100];
  BigInt *a=bnNewBigInt(50, 0);
  char *input="-123456789000";
  bnStrToInt(a, input);
  bnIntToStr(ans,a);
  ```

After executing this code the value of the array pointed by `ans` will be `"-123456789000"`.

1.2 Arithmetic Operations

1.2.1 Addition

For adding two big integers you can use the following function:

```
void bnAddInt(BigInt* res, BigInt* sum1, BigInt* sum2)
```

Purpose: To compute the sum of two big integers, `res` = `sum1` + `sum2`.

Arguments:

- **res:**

  Pointer to the big integer where the result of adding `sum1` and `sum2` will be saved, remember that the size of `res` must be enough for holding the result.

- **sum1:**

  Pointer to the big integer which is the first operand.
```

- **sum2:**

  Pointer to the big integer which is the second operand.

Return Value :

None.

Example:

- ```
  char array[100];
  BigInt *a=bnNewBigInt(50, 0);
  BigInt *b=bnNewBigInt(50, 0);
  BigInt *ans=bnNewBigInt(50, 0);
  bnStrToInt(a, "2000000000");
  bnStrToInt(b,"3000000001");
  bnAddInt(ans,a,b);
  bnIntToStr(array,ans);
  ```

After this code is executed the value of the big integer `ans` will be the sum of `a` and `b`, and the value of the array pointed by `array` will be `"5000000001"`.

1.2.2 Subtraction

For subtracting two big integers you can use the following function:

```
void bnSubInt(BigInt *res, BigInt *a, BigInt *b);
```

Purpose: To compute the subtraction of two big integers, `res = a - b`.

Arguments:

- **res:**

  Pointer to the big integer where the result of computing `a` minus `b` will be saved.

- **a:**

  Pointer to the big integer which is the minuend.

- **b:**

  Pointer to the big integer which is the subtrahend.

Return Value :

None.
```

Example:

```
char array[100];
BigInt *a=bnNewBigInt(50, 0);
BigInt *b=bnNewBigInt(50, 0);
BigInt *ans=bnNewBigInt(50, 0);
bnStrToInt(a, "24545646464464789789746");
bnStrToInt(b,"387979789746464679");
bnSubInt(ans,a,b);
bnIntToStr(array,ans);
```

After this code is executed the value of the big integer `ans` will be the result of computing `a` minus `b`, and the value of the array pointed by `array` will be `"24545258484675043325067"`.

1.2.3 Multiplication

For computing the multiplication of two big integers you can use the following function:

```
void bnMulInt(BigInt *res, BigInt *a, BigInt *b);
```

Purpose: To multiply two big integers, `res` = `a` * `b` .

Arguments:

- **res:**

  Pointer to the big integer where the result of multiplying `a` and `b` will be saved.

- **a:**

  Pointer to the big integer which represents one factor.

- **b:**

  Pointer to the big integer which represents one factor.

Return Value :

None.

Example:

```
char array[100];
BigInt *a=bnNewBigInt(50, 0);
BigInt *b=bnNewBigInt(50, 0);
```

```
BigInt *ans=bnNewBigInt(50, 0);
bnStrToInt(a, "123456789");
bnStrToInt(b,"987654321");
bnMulInt(ans,a,b);
bnIntToStr(array,ans);
```

After this code is executed the value of the big integer `ans` will be the result of multiplying two big integers (`a` and `b`), and the value of the array pointed by `array` will be `"121932631112635269"`.

1.2.4 Division

For computing the integer division of two big integers you can use the following function:

```
void bnDivInt(BigInt *ans, BigInt *a, BigInt *b, BigInt *res);
```

Purpose: To make an integer division of two big integers, $ans = \lfloor \frac{a}{b} \rfloor$, $res = a - ans * b$.

Arguments:

- **ans:**

  Pointer to the big integer where the quotient of dividing `a` by `b` will be saved.

- **a:**

  Pointer to the big integer which represents the dividend.

- **b:**

  Pointer to the big integer which represents th divisor.

- **res:**

  Pointer to the big integer where the remainder of `a` divided by `b` will be saved.

Return Value :

None.

Example:

```
char quotient[100];
char remainder[100];
BigInt *ans=bnNewBigInt(50, 0);
BigInt *a=bnNewBigInt(50, 0);
BigInt *b=bnNewBigInt(50, 0);
BigInt *rem=bnNewBigInt(50, 0);
char *dividend="13213131231314644665467879";
char *divisor="9987999784543";
bnStrToInt(a,dividend);
bnStrToInt(b,divisor);
bnDivInt(ans, a, b, rem);
bnIntToStr(quotient,ans);
bnIntToStr(remainder,rem);
printf("Quotient:%s Remainder:%s\n",quotient,remainder);
```

The following line will be printed to standard output when this code is executed:

```
Quotient:1322900632393 Remainder:1562218966480
```

1.2.5 Exponentiation

For computing exponentiation of integer numbers you can use the following function:

```
void bnPowInt(BigInt *ans, BigInt *a, int b);
```

Purpose: To compute the result of raising an integer number to another integer number, $ans = a^b$.

Arguments:

- **ans:**

  Pointer to the big integer where the result of raising `a` to the power of `b` will be saved. Make sure that this big integer has enough size for holding the exponentiation result.

- **a:**

  Pointer to the big integer which represents the exponentiation base.

- **b:**

  Pointer to the integer which represents the exponent. Notice that the type of this argument is `int` instead of `BigInt`, because it does not make sense to compute the exponentiation with very large powers since this kind of operation exceeds the computer capabilities.

Return Value :

None.

Example:

- ```
  char array[100];
  BigInt *ans=bnNewBigInt(50, 0);
  BigInt *a=bnNewBigInt(50, 2);
  bnPowInt(ans, a, 256);
  bnIntToStr(array,ans);
  ```

The value of the string pointed by `array` after this code is executed will be:
`"115792089237316195423570985008687907853269984665640564039457584007913129639936"`
which is 2^{256}.

1.2.6 Modular exponentiation

For computing modular exponentiation you can use the following function:

```
void bnPowModInt(BigInt *ans, BigInt *a, BigInt* b, BigInt *mod);
```

Purpose: To compute the result of an exponentiation modulo an integer number, $ans \equiv a^b$ *modulo* mod.

Arguments:

- **ans:**

  Pointer to the big integer where the result of raising `a` to the power of `b` module `mod` will be saved.

- **a:**

  Pointer to the big integer which represents the exponentiation base.

- **b:**

  Pointer to the big integer which represents the exponent. Notice that in contrast with the function `bnPowInt` this exponent can be a large number because this is modular exponentiation.

Return Value :

None.
```

Example:

- ```
  char array[100];
  char *exp="123123131312312331312131456";
  BigInt *ans=bnNewBigInt(50, 0);
  BigInt *a=bnNewBigInt(50, 45646464);
  BigInt *b=bnNewBigInt(50, 0);
  bnStrToInt(b,exp);
  BigInt *mod=bnNewBigInt(50, 1000003);
  bnPowModInt(ans, a, b, mod);
  bnIntToStr(array,ans);
  ```

  The value of the string pointed by `array` will be `"506178"` which is $45646464^{123123131312312331312131456} \ mod\ 1000003$

1.3 Other Operations

1.3.1 Logical Left Shift

For shifting all bits of a big integer number to the left you can use the following function:

```
void bnShiftLBits(BigInt* res, BigInt* a, unsigned int bits)
```

Notice that this shift is made over binary digits instead of 2^B digits, where B is the bitlength of an `int` data type. Remember that this operation has the effect of multiplying by 2^{bits} the number which is being shifted, as a technical detail this shift is not circular, so the bit positions which get empty are filled with zeros.

Purpose: To shift to the left all bits of a big integer `bits` positions, $res = a << bits$.

Arguments:

- **res:**

  Pointer to the big integer where the result of shifting `a` to the left a number of positions given by `bits` will be saved.

- **a:**

  Pointer to the big integer which will be shifted to the left.

- **bits:**

  Is the number of bit positions the big integer `a` will be shifted to the left.

Return Value :

None.
```

Example:

- ```
  char array[100];
  BigInt *res=bnNewBigInt(50, 0);
  BigInt *a=bnNewBigInt(50, 1000000000);
  bnShiftLBits(res,a,100);
  bnIntToStr(array,res);
  ```

The value of the string pointed by `array` will be `"1267650600228229401496703205376000000000"` which is $1000000000 * 2^{100}$.

1.3.2 Logical Right Shift

For shifting all bits of a big integer number to the right you can use the following function:

```
void bnShiftRBits(BigInt *res, BigInt *a, unsigned int bits);
```

Notice that this shift is made over binary digits instead of 2^B digits, where B is the bitlength of an `int` data type. Remember that this operation has the effect of making an integer division of the number which is being shifted by 2^{bits} , as a technical detail this shift is not circular, so the bit positions which get empty are filled with zeros.

Purpose: To shift to the right all bits of a big integer `bits` positions, $res = a >> bits$.

Arguments:

- **res:**

  Pointer to the big integer where the result of shifting `a` to the right a number of positions given by `bits` will be saved.

- **a:**

  Pointer to the big integer which will be shifted to the right.

- **bits:**

  Is the number of bit positions the big integer `a` will be shifted to the right.

Return Value :

None.
```

Example:

- ```
  char ans[100];
  char *number="213131236467979794614567946321324 59";
  BigInt *res=bnNewBigInt(50, 0);
  BigInt *a=bnNewBigInt(50, 0);
  bnStrToInt(a, number);
  bnShiftRBits(res,a,100);
  bnIntToStr(ans,res);
  ```

The value of the character array pointed by ans will be `"16813"` which is $\lfloor 213131236467979794614567946321324 59/2^{100} \rfloor$.
  ```

Chapter 2

Putting all together

The aim of this chapter is to give a few examples about how to use all the BigNum primitives for implementing an specific algorithm, so the BigNum user gets closer to the way of working with BigNum library and he can take advantage of all the features of this library.

Chapter 3

Intern Structure

3.1 Addition and subtraction

For addition and subtraction the library uses the schoolbook algorithms but in base b. To support signed integer addition and subtraction, the library has methods implemented for unsigned addition and subtraction. The function for unsigned addition is:

```
bnUAddInt(BigInt *res, BigInt *a, BigInt *b)
```

The function receives two integers `a` and `b` and adds them ignoring the signs in `res`. Let a_i and b_i be the digit i-th of the first and second operands respectively, and c_i be the i-th digit of the result, then the internal behavior of this function is given by:

$$c_i = a_i + b_i + k$$

where k is the carry digit of the operation on the $(i-1)$-th digit. The function for unsigned subtraction is:

```
bnUSubInt(BigInt *res, BigInt *a, BigInt *b)
```

The function receives two integers `a` and `b` and subtracts them in `res` as with natural numbers, the magnitude of `a` must be greater or equal to `b` to avoid a less than zero result. Let a_i and b_i be the digit i-th of the first and second operands respectively, and c_i be the i-th digit of the result, then the internal behavior of this function is given by:

$$c_i = a_i - b_i - k$$

where k is the borrow digit of the operation on the $(i-1)$-th digit.

With the operations for addition and subtraction on natural numbers implemented, is easy to implement a signed addition with the following algorithm:

- If the signs of both operands are equal, then use unsigned addition and put the sign of the operands to the results
- If the signs are different, then use unsigned subtraction to subtract the operand of lowest magnitude from the operand of highest magnitude and keep the sign of the operand of highest magnitude.

With the signed addition implemented, the signed subtraction was implemented by changing the sign of the second operand and calling the signed addition. The signature of the functions for signed addition and subtraction was explained in the chapter 1.

3.1.1 Complexity

Let n be the number of digits in base 2^B of each operand. As both addition and subtraction make a single loop on the number of digits its complexity is $O(n)$.

3.2 Multiplication

For multiplication the library also uses the schoolbook algorithm over base 2^B. When someone wants to multiply numbers of several digits in base 10, the only thing that person needs to know is how to multiply single digits (Using the multiplication table for example) and also how to add numbers of several digits. For this algorithm we use the fact that the processor already knows how to multiply two numbers of a single digit in base 2^B and the algorithm of section 3.1 to add numbers of several digits in base 2^B. The algorithm to multiply two big integers a and b require two temporal big integer numbers (`tmp` and `sum`) to store the result of multiplying all a by the digit i of b and the sum of all partial multiplication respectively, the algorithm works as follows:

- set `sum` to zero
- for each digit b_i of b:
  - multiply the operand a by the digit b_i and store it in `tmp`
  - accumulate `tmp` shifted i digits to the left in `sum` (`sum = sum + digitShift(tmp,i)`)

The previous algorithm uses a method for multiplying a full big integer by a single digit in base 2^B, the method `bnMulIntWord(BigInt* res, BigInt *a, bn_word b)` were implemented for that purpose. The implementation for this method also uses the schoolbook algorithm, taking into account that the multiplication of two single digits never overflows a two digit variable the algorithm multiplies each digit of a by the digit b and stores the result in a variable m of type `bn_dword` and $2B$ bits capacity. Then the least significant digit in base 2^B of m is stored in the resulting integer and the other digit of m is taken as the carry for the next operation.

3.2.1 Complexity

Let n be the number of digits in base 2^B of the operands a and b. The complexity of `bnMulIntWord` would be $O(n)$, and the complexity of `bnMulInt` would be $O(n^2)$.

3.3 Division

3.4 Exponentiation and modular exponentiation