

Table of Contents

- 1. [Foreword & Introduction](#)
- 2. [Introduction to Base 10 Implementation](#)
- 3. [Creating big integer objects](#)
 - 3.1 [Creating from digit arrays](#)
 - 3.2 [Creating from unsigned numbers](#)
 - 3.3 [Creating from strings](#)
 - 3.4 [Creating from hexadecimal string](#)
- 4. [Arithmetic operations](#)
 - 4.1 [Addition](#)
 - 4.2 [Subtraction](#)
 - 4.3 [Simple multiplication](#)
 - 4.4 [Better multiplication](#)
 - 4.5 [Division](#)
 - 4.6 [Exponentiation](#)
 - 4.7 [Modular exponentiation \(ModPow\)](#)
 - 4.8 [Shifting](#)
 - 4.9 [Comparison](#)
 - 4.10 [Parity \(Even / Odd\)](#)
- 5. [Extending to other bases](#)
- 6. [Conclusion & References](#)

[Download C source code](#) (59 kb)

1. Foreword & Introduction

[\[Back to top\]](#)

This article explains a very simple (and inefficient) implementation of an (unsigned) big integer library. While there are many open-source implementations of big integer libraries freely available on the internet, nearly all of them lack an explanation how they work internally.

Actually I could not find any useful basic information about the algorithms used to store big integers and implement arithmetic operations. Indeed, there are many theoretical articles and books available about so-called arbitrary precision arithmetic (e.g. in [\[5\]](#)), but they are quite complex to understand and not feasible as a practical starting point.

So I thought why not trying myself writing a big integer library and share my knowledge with you :-).

[\[Back to top\]](#)

2. Introduction to Base 10 Implementation

The easiest approach is to store the single digits of a number in an array with base 10.

For example, the number 5831 can be written in base 10 positional notation as $5 * 10^3 + 8 * 10^2 + 3 * 10^1 + 1 * 10^0$.

The usage of an array of bytes seems a feasible approach. Based on this idea, following a concrete implementation specification:

- The single digits are stored in base 10 in an array of bytes.
- The least significant digit is stored at position 0, the second least significant digit at position 1, and so on until the most significant digit is stored at the highest position.
- There are no leading zeros stored. Thus the length of the array is always equal to the amount of digits of a number.
- To avoid resize of arrays, the class is implemented immutable – once a bigint number is created, it is not changed anymore.
- This results in our example implementation into a class `SunBigUInt10` that use a single class member: `private byte[] digits;`

For example, the value 5831 would be stored as `byte[] digits = { 1, 3, 8, 5 }`. Note the reversed order of digits compared to the natural way.

[\[Back to top\]](#)

3. Creating big integer objects

Well, at first it's required to create big integers with the desired value from scratch. There are different possible ways for that.

[\[Back to top\]](#)

3.1 Creating from digit arrays

A simplistic approach is to directly create the array with the digits manually. This works but is cumbersome.

Additionally, it is still required to fulfill the requirements defined in [chapter 2](#), otherwise subsequent operations will fail. So it must be ensured that leading zeros are removed before creating a copy of the array. As the implementation is very easy, the listing is left out, but is contained in the source code.

[\[Back to top\]](#)

3.2 Creating from unsigned numbers

A more comfortable way is creating unsigned big integer objects from unsigned integers like `uint` or `ulong`.

Following algorithm describes one way to fulfill this task. Note that integer divisions are used, no floating point division.

- Get the number of digits of the provided number by dividing it by 10 until it is zero and count the number of divisions. This information might be required to create the array.
- To get the single digits, divide the provided number by 10 until it is zero and store the remainder in each step.

Example to create a big integer object from 5831:

Optionally, at first get the number of digits:

```
5831 / 10 = 583. Division number = 1.
583 / 10 = 58. Division number = 2.
58 / 10 = 5. Division number = 3.
5 / 10 = 0. Division number = 4. Value is zero, so algorithm terminates and the number of
digits is 4.
```

The same divisions are performed to get the particular single digits, each remainder is a single digit of the number.

```
5831 / 10 = 583, remainder 1. Store 1 as first digit: digits[0] = 1.
583 / 10 = 58, remainder 3. Store 3 as next digit: digits[1] = 3.
58 / 10 = 5, remainder 8. Store 8 as next digit: digits[2] = 8.
5 / 10 = 0, remainder 5. Store 5 as last digit: digits[3] = 5. Finally, the digits array contains the
result.
```

A remarkable limitation in creating big integers from native unsigned integers is that only numbers in range of the supported unsigned integers can be created and not arbitrary long ones as initially intended.

Following a sample implementation:

```
public static SunBigUInt10 FromLong(ulong number)
{
    if (number < 0) throw new ArgumentOutOfRangeException("Negative numbers not supported.");
    SunBigUInt10 n = new SunBigUInt10();
    if (number == 0) // special shortcut handling for zero
    {
        n.digits = new byte[1];
        return n;
    }
    // count number of digits
```

```

ulong numberCopy = number;
int numDigits = 0;
while (numberCopy > 0)
{
    numberCopy /= 10;
    numDigits++;
}
// create array and fill it with the digits of the number
n.digits = new byte[numDigits];
int i = 0;
while (number > 0)
{
    n.digits[i] = (byte)(number % 10);
    number /= 10;
    i++;
}

return n;
}

```

[\[Back to top\]](#)

3.3 Creating from strings

Creating big integers from strings finally allows to create big integers with an insane number of digits – only the available memory limits the size of the numbers.

Based on the fact that each character of the string represents a single digit, just iterate over each character of the string and convert it to a digit. This turns out surprisingly easy because the ASCII code of digit '0' is 0x30, '1' is 0x31 ... and '9' is 0x39. So by subtracting 0x30 from a single ASCII character, the decimal digit can be retrieved.

Example for "5831":

The string "5831" has four characters. '5' has ASCII code 0x35, '8' has 0x38, '3' has 0x33 and '1' has 0x31.

The least significant digit is '1', so store 0x31 – 0x30 in digits[0]. Proceeding with the next digit '3' results in digits[1] = 0x33 – 0x30 = 3, and so on.

Following table shows a sample implementation of the most important part, leaving out parameter checks and removing white spaces from the string:

```

public static SunBigUInt10 FromString(string number)
{
    SunBigUInt10 n = new SunBigUInt10();
    n.digits = new byte[number.Length];
    for (int i = 0; i < number.Length; i++)
    {
        if (!Char.IsDigit(number[number.Length - i - 1]))
        {
            throw new ArgumentException("Invalid decimal digit: " + number[number.Length - i - 1]);
        }
        n.digits[i] = (byte)(number[number.Length - i - 1] - 0x30);
    }
    return n;
}

```

When iterating over the string, take care of the reversed order, i.e. that the least significant digit is the last character in the string but needs to be stored at first position in the digits array.

[\[Back to top\]](#)

3.4 Creating from hexadecimal strings

It might become necessary to create big integer objects from string in hexadecimal notion like e.g. "0x3A". This is a bit more complex than the decimal case in previous chapter, as here a single string character cannot necessarily be represented also as a single digit. For example, the single character 'C' stands for decimal number 12 which has two digits.

At first, consider in general the conversion of hexadecimal number to a decimal number. A hexadecimal number can also be written in positional notation, just with base 16. As a numerical system with base 16 uses also 16 different digits, beside the digits in range [0, 9], the characters in range [A, F] are interpreted as digits 10, 11, .. , 15.

Example - Converting 0x2FC

The value 0x2FA is written in base 16 positional notation as $2 * 16^2 + 15 * 16^1 + 12 * 16^0$.

This is $2 * 256 + 15 * 16 + 12 = 764$.

Note that this can also be written as $2 * 16 * 16 + 15 * 16 + 12$ which is closer to the following algorithm.

The algorithm is implemented as follows where n is a big integer object:

- Init n = 0
- For each character digit in string, starting from most significant one:
 - $n = n * 16 + \text{value of character digit}$

Example - Processing 0x2FC

Reminder: '2' has value 2, 'F' has value 15 and 'C' has value 12.

```
n = 0;
n = n * 16 + '2' = 0 * 16 + 2 = 2;
n = n * 16 + 'F' = 2 * 16 + 15 = 47;
n = n * 16 + 'C' = 47 * 16 + 12 = 764
```

The following sample implementation shows the important part of the algorithm, leaving out the removal of the "0x" prefix. Especially note the conversion of the letter [A,F] where first 'A' is subtracted to map [A,F] to [0, 5], then adding 10 to get the requested values [10, 15]. The same is done to support lower case letters.

```
public static SunBigUInt10 FromHexString(string number)
{
    SunBigUInt10 n = SunBigUInt10.FromLong(0);
    for (int i = 0; i < number.Length; i++)
    {
        char c = number[i];
        int digitVal;

        if ((c >= '0') && (c <= '9'))
        {
            digitVal = c - '0';
        }
        else if ((c >= 'a') && (c <= 'f'))
        {
            digitVal = c - 'a' + 10;
        }
        else if ((c >= 'A') && (c <= 'F'))
        {
            digitVal = c - 'A' + 10;
        }
        else
        {
            throw new ArgumentException("Invalid hexadecimal digit: " + c);
        }

        n = (n * 16) + digitVal;
    }
    return n;
}
```

But stop! In third last line, multiplication and addition is used – for our big integer objects.

Yes, that's right, currently these two arithmetic operations are required for the algorithms and have not been addressed yet.

They will be discussed later in subsequent chapters.

[\[Back to top\]](#)

4. Arithmetic operations

Being able to create big integer objects (that consists actually only of an array of digits), now it's time to present operations on those objects. Be prepared to be confronted with primary school mathematics!

[\[Back to top\]](#)

4.1 Addition

The addition with the big integer objects works exactly like column addition in school:

1. Write both numbers among each other, aligned right on the least significant.
2. Starting with the least significant digits, add the digit of both numbers and write the result below. If the result is equal or greater 10, take a carry of 1 two the next column.
3. Repeat previous step for each column of digits.

Addition example 1 - Adding 46 and 28

Write both numbers among each other and start adding the digits from the right:

```

46
28
--
74

```

Step 1: $6 + 8$ gives 14. This is equal or greater than 10, so subtract 10 from 14 and take a carry of 1 to the next column. First digit is: $14 - 10 = 4$, carry is 1.

Step 2: $4 + 2$ is 6. Adding the carry gives $6 + 1 = 7$. 7 is less than 10, do not subtract, carry is 0. Second digit is 7, no carry is left.

Carry is 0 and it's the last column -> finished. The result is 74.

Addition example 2 - Adding 89 and 31

```

89
31
--
120

```

Step 1: $9 + 1 = 10$. Subtract 10. First digit is 0, carry is 1.

Step 2: $8 + 3 = 11$. Current carry is 1, so add $11 + 1 = 12$. Second digit is 2, carry is 1.

Step 3: No column left (this can also be considered as there are zero in the columns), carry is 1. Third digit is 1, carry is 0. The result is 120.

Addition example 3 - Adding 131 and 1

If the numbers have different length, this can be treated as the shorter number is prefixed with zeros until both number have equal length:

```

131
001
---
132

```

This leads to some useful observations:

- The addition of a number of n digits with a number of m digits always either result in a number with $\max(n,m)$ digits or $\max(n,m) + 1$ digits, as there is maximum one carry.
Therefore the result array should be allocated with $n + m + 1$ digits, and afterwards trimmed if necessary to not include leading zeros.
- Both number may have equal length or not. If not, all columns of the longer number must be processed, if required assuming a 0 for a non-existing column of the shorter number.

This could lead to following possible implementation:

```

public static SunBigUInt10 operator +(SunBigUInt10 a, SunBigUInt10 b)
{
    /* result array with n + m + 1 digits */
    byte[] temp = new byte[Math.Max(a.digits.Length, b.digits.Length) + 1];
    byte carry = 0;
    for (int i = 0; i < temp.Length; i++)
    {
        /* get the digits of both numbers of current column, use 0 if column of number does not
        exist */
        int anum = i < a.digits.Length ? a.digits[i] : 0;
        int bnum = i < b.digits.Length ? b.digits[i] : 0;
        byte sumNum = (byte)(anum + bnum + carry);
        if (sumNum >= 10)
        {
            sumNum -= 10;
            carry = 1;
        }
    }
}

```

```

        else
        {
            carry = 0;
        }
        temp[i] = sumNum;
    }
    return new SunBigUInt10(temp); /* this will trim a possible leading zero from temp array
before creating the object */
}

```

[\[Back to top\]](#)

4.2 Subtraction

Based on the addition algorithm in previous chapter, the subtraction is nearly much the same, with following differences:

- Instead of adding the single digits of a column, they are subtracted.
- If the lower digit is greater than the digit above, thus the result is smaller than zero, add 10 and then take a carry of one to the next column

An example will make it clearer:

Subtraction example - Subtracting 99 from 123

If the numbers have different length, this can be treated as the shorter number is prefixed with zeros until both number have equal length:

```

123
 99
---
24

```

Step 1: $3 - 9$ is -6 . -6 is smaller than 0, so adding 10 gives $-6 + 10 = 4$, saving a carry. First digit is 4, carry is 1.

Step 2: $2 - 9$ is -7 . Subtracting the carry finally gives $-7 - 1 = -8$. -8 is smaller than 0, so adding 10 gives $-8 + 10 = 2$, saving a carry. Second digit is 2, carry is 1.

Step 3: $1 - 0$ is 1. Subtracting the carry finally gives $1 - 1 = 0$. Third digit is 0, carry is 0.

The result is 024, thus 24 by removing leading zeros.

It's worth to remind at this point that an **unsigned** big integer library is implemented.

Negative numbers are not supported, it must be ensured that the first number a is greater than the second number b , otherwise the result is meaningless.

This implies that the number of digits of a is equal or greater than the number of digits of b . Further, the number of digits of the result is not greater than the number of digits of a .

A possible implementation could look like following:

```

public static SunBigUInt10 operator -(SunBigUInt10 a, SunBigUInt10 b)
{
    if (b > a)
    {
        throw new OverflowException("b is greater than a.");
    }

    byte[] temp = new byte[a.digits.Length];
    byte borrow = 0;
    for (int i = 0; i < temp.Length; i++)
    {
        int anum = a.digits[i];
        int bnum = i < b.digits.Length ? b.digits[i] : 0;
        byte diffNum;
        if (anum < (bnum + borrow))
        {
            // overflow
            diffNum = (byte)(anum + 10 - bnum - borrow);
            borrow = 1;
        }
        else
        {
            diffNum = (byte)(anum - bnum - borrow);
            borrow = 0;
        }
        temp[i] = diffNum;
    }

    return new SunBigUInt10(temp);
}

```

4.3 Simple multiplication

A simple approach to implement multiplication is to follow the way as we learned it in school. At first an easy example to get "comfortable" with multiplication, which uses the gained knowledge from addition:

Multiplication example 1 - Multiply $a = 45$, $b = 5$

```
45 * 6
-----
270
```

Step 1: Multiply 6 and 5 = 30. This is greater or equal to 10, take the least significant digit 0 and save carry = 3. First digit is 0, carry is 3.

Step 2: Multiply 6 and 4 gives 24, adding the carry results in $24 + 3 = 27$. This is greater or equal to 10, take the least significant digit 7 and save carry = 2. Second digit is 7, carry is 1.

Step 3: Imagine writing 45 as 045, then multiply 5 and 0 gives 0, adding carry result in 1. No subtraction, no carry, third digit is 2.

The result is 270.

Following points can be observed from example 1:

- The digit of b is multiplied with each digit of a , moving from right to left.
- Similar to addition, if the product is smaller than 10, take the digit and move on.
If the product is greater or equal to 10, take the least significant digit and save the other part of the number as carry which is added to next product.
- Contrary to addition, the carry is not only 0 or 1, but can be greater (in range $[0, 9]$).

Now this will be extended for the case that also b has more than one digit so that each digit of b is multiplied with each digit of a .

Multiplication example 2 - Multiply $a = 36$, $b = 184$

```
36 * 184
-----
144
288
36
-----
6624
```

Row 1. $4 * 6 = 24$. Digit is 4, store 2. $4 * 3 + 2 = 14$. Digit is 4, store 1. $4 * 0 + 1 = 1$. Digit is 1. First row is 144.

Row 2. $8 * 6 = 48$. Digit is 8, store 4. $8 * 3 + 4 = 28$. Digit is 8, store 2. $1 * 0 + 2 = 2$. Digit is 2. Second row is 288.

Row 3. $1 * 6 = 6$. Digit is 6, store 0. $1 * 3 + 0 = 3$. Digit is 3, store 0. Result is 36.

Note that the result of each row is shifted one more column to the left. 144 is shifted 0 columns to the left, 288 is shifted 1 column to the left and 36 even 2 columns.

A left shift by n columns is the same as multiplying by 10^n . So the result of each row is actually multiplied by 10^n :

$$144 * 10^0 = 144 * 1 = 144.$$

$$288 * 10^1 = 288 * 10 = 2880.$$

$$36 * 10^2 = 36 * 100 = 3600.$$

Finally all values are added to get the actual result: $144 + 2880 + 3600 = 6624$.

This could lead to following algorithm:

- result = 0;
- Iterate over each digit of b from right to left, call it $bDigit$.
 - rowProduct = 0;

- Iterate over each digit of a from right to left.
 - calculate the row product with $bDigit$
- Multiply sum of row n with 10^n
- add multiplied result to total result

There is one important point here: For the product of two single digits, primitive types can be used as the maximum value could be $9 * 9 + 9 = 90$. This is good news as we need division by 10 and modulo by 10 for this calculation.

But note that the two factors a and b could have arbitrary length, thus also the result variable as well as the rowProduct variable must be big integer objects.

But wait, for result and rowProduct variables, beside addition we need to multiply by 10^n – and multiplication is not yet defined for big integer objects! So what to do here?

Fortunately, there is a trick here: Multiplying a big integer object by 10^n is the same as left shifting it by n positions to the left because base 10 is used.

This is similar to left shifting a primitive integer i objects by n resulting in $i * 2^n$ because they are stored in base 2.

Example

The number 1385, stored as `byte[] digits = { 1, 3, 8, 5 }`, left shifted by 2 positions, results in `byte[] digits = { 1, 3, 8, 5, 0, 0 }`. This is equal to multiply 1385 by $10^2 = 1385 * 100 = 138500$.

Putting all this together leads to following simple and inefficient multiplication implementation:

```
public static SunBigUInt10 MulSimple(SunBigUInt10 a, SunBigUInt10 b)
{
    SunBigUInt10 result = 0;
    uint bStep = 0;
    for (int bIdx = 0; bIdx < b.digits.Length; bIdx++)
    {
        byte bDigit = b.digits[bIdx];
        SunBigUInt10 rowProduct = 0;
        byte carry = 0;
        uint aStep = 0;
        for (int aIdx = 0; aIdx < a.digits.Length; aIdx++)
        {
            byte aDigit = a.digits[aIdx];
            /* calculate multiplication of two single digits from a and b: a[i] * b[j] + carry */
            byte digitMulResult = (byte)((bDigit * aDigit) + carry);

            if (aIdx < a.digits.Length - 1)
            {
                /* not last multiplication of current row, store digit in sum of current row,
                update carry */
                SunBigUInt10 digitSumShifted = SunBigUInt10.FromLong((ulong)(digitMulResult %
                10)).ShiftLeftDigits(aStep);
                rowProduct += digitSumShifted;
                carry = (byte)(digitMulResult / 10);
                aStep++;
            }
            else
            {
                /* last step takes into account that no leading zero is appended */
                SunBigUInt10 digitSumShifted = SunBigUInt10.FromLong((ulong)
                (digitMulResult)).ShiftLeftDigits(aStep);
                rowProduct += digitSumShifted;
            }
            /* multiply sum of row n with 10^n by left shifting by n */
            SunBigUInt10 rowProductSumShifted = rowProduct.ShiftLeftDigits(bStep);
            /* add rowproduct to total result */
            result += rowProductSumShifted;
            bStep++;
        }
    }
    return result;
}
```

[\[Back to top\]](#)

4.4 Better multiplication

The multiplication algorithm presented in the previous chapter is more complicated than required. Analysis reveals that the row products actually do not need to be calculated explicitly nor the shifting by base 10 is required because each row product can be added to the result "on the fly".

Using the previous example of $36 * 184$ and starting with a result value of 0, each digit of first row product $36 * 4 = 144$ is added to each digit of the result value using normal addition with carry as presented in chapter 4.1. For the second row product $36 * 8$, each digit of this row product is added to each digit of the result value - however not starting at digit 0 of the result value, but at digit 1 of the result, or more generally at digit $n+1$ for row product n (this corresponds to the previous left shifting by 10^n).

(Better) multiplication - Multiply $a = 36$, $b = 184$

Start with result = 0. It is written as 00000 (will be discussed below) where the single digits can be accessed by result[0] (least significant digit) to result[4] (most significant digit).

Row 0: $36 * 4$. Set carry = 0.

$6 * 4 + \text{result}[0 + 0] + \text{carry} = 6 * 4 + 0 + 0 = 24$. Set first digit 4 as result[0] = 4 => Result is 0004. Carry = 2.

$3 * 4 + \text{result}[0 + 1] + \text{carry} = 3 * 4 + 0 + 2 = 14$. Set first digit 4 as result[1] = 4 => Result is 0004. Carry = 1.

Row finished, but carry is not 0. So add carry to result[0 + 2]: $1 + 0 = 1$. Result is 0144

Row 1: $36 * 8$. Set carry = 0.

$6 * 8 + \text{result}[1 + 0] + \text{carry} = 6 * 8 + 4 + 0 = 52$. Set first digit 2 as result[1] = 2 => Result is 0124. Carry = 5.

$3 * 8 + \text{result}[1 + 1] + \text{carry} = 3 * 8 + 1 + 5 = 30$. Set first digit 0 as result[2] = 0 => Result is 0024. Carry = 3.

Row finished, but carry is not 0. So add carry to result[1 + 2]: $0 + 3 = 3$. Result is 3024

Note that this is $144 + 288 * 10 = 3024$ - same value compared to second step of simple multiplication.

Row 2: $36 * 1$. Set carry = 0.

$6 * 1 + \text{result}[2 + 0] + \text{carry} = 6 * 1 + 0 + 0 = 6$. Set first digit 6 as result[2] = 6 => Result is 3624. Carry = 0.

$3 * 1 + \text{result}[2 + 1] + \text{carry} = 3 * 1 + 3 + 0 = 6$. Set first digit 6 as result[3] = 6 => Result is 6624. Carry = 0.

Row finished, carry is 0, nothing more to do => Result is 6624.

Points worth to mention:

- The index of the result digit to use for calculation is the sum of the **current zero-based row index** and the **index of digit of factor a**.
- Q: How many digits has the result number? A: The multiplication of a number of n digits with a number of m digits either result in a number with $n + m$ or $n + m + 1$ digits.
As the algorithm accesses the digits of the result value in range $[0, n+m]$ respectively $[0, n + m + 1]$, the implementation must take special care to avoid out-of-bounds accesses. The chosen, most probably easiest way, is to create a result variable (with big integer type of course) with the maximum possible digits, initialized with 0.

Following a sample implementation for the multiplication:

```
public static SunBigUInt10 Mul(SunBigUInt10 a, SunBigUInt10 b)
{
    byte[] tmpArray = new byte[a.digits.Length + b.digits.Length];

    int i, j;
    for (j = 0; j < b.digits.Length; j++)
    {
        byte carry = 0;
        for (i = 0; i < a.digits.Length; i++)
        {
            carry += (byte)(a.digits[i] * b.digits[j] + tmpArray[i + j]);
            tmpArray[i + j] = (byte)(carry % 10);
            carry /= 10;
        }

        if (carry != 0)
        {
            tmpArray[i + j] = (byte)(carry % 10);
        }
    }

    return new SunBigUInt10(tmpArray);
}
```

4.5 Division

Division for arbitrary precision arithmetic numbers is by far the most complicated arithmetic operation. There are very complex algorithms described in literature which are hard to understand and which would require own tutorials itself to explain. As the goal of this article is to implement a working, but simple (at the cost of efficiency) big unsigned integer, here the probably simplest algorithm for division is presented. It's a recursive algorithm, but first start with the basics.

Problem definition: The aim is to calculate the quotient and remainder of the division of the two numbers a (dividend) and b (divisor):

The quotient q is defined as the result of division: $q = a / b$.

The remainder r is defined as the result of the modulo operation: $r = a \% b$.

This can also be summarized and written as $a = q * b + r$.

Algorithm: The algorithm distinguishes three cases.

Case 1: If $a < b$, it's trivial, because $q = 0$ and $r = a$.

The main idea of the algorithm is to recursively multiply the divisor b by 2 until the trivial case is reached.: $a = q * b + r$
 $\Rightarrow a = q' * 2 * b + r'$.

Then traverse the taken recursive way back to the root and calculate q' and r' each step to finally gain q and r .

The next two cases handle the general cases. The distinction is based on the requirement that the remainder r is always smaller than the divisor b : Constraint: $r < b$.

Case 2: The doubling of the divisor b **does not violate** constraint $r < b$.

Assume from $a = q * b + r$, the step to $a = q' * 2 * b + r'$ has been done and the values q' and r' have been calculated. Now the step back shall be taken to retrieve q and r . If r' is smaller than b , then the remainder remains the same and q is doubled: $q = 2 * q'$ and $r = r'$.

Example for division - Case 2

Division for $a = 100$, $b = 40$ shall be calculated. The result for $100 / 80$ is known which is $q' = 1$ and $r' = 20$.

To reach the previous step, divisor of previous step 40 is compared to remainder r' : The constraint is not violated as $20 < 40$.

Therefore $q = q' * 2 = 1 * 2 = 2$ and remainder $r = r' = 20$ which is correct as $100 / 40$ results in $q = 2$, remainder 20.

The same written in another notation:

$$100 = q * 40 + r.$$

$$100 = q' * 80 + r'.$$

$$\rightarrow 100 = 1 * 80 + 20. \text{ As } 20 < 40:$$

$$100 = 2 * q' * 40 + r = 2 * 1 * 40 + 20.$$

Case 3: The doubling of the divisor b **does violate** constraint $r < b$, i.e $b \geq r$.

Again, the step from $a = q' * 2 * b + r'$ (where the solution is known) back to $a = q * b + r$ shall be taken.

However, the remainder r' is equal or greater than b .

This is invalid as the remainder must be smaller than the divisor - otherwise the quotient could be increased and the remainder decreased until the constraint is fulfilled again. This is exactly what will be done: $q = 2 * q' + 1$ and $r = r' - b$.

Note that it cannot be the case that the divisor b fits more than once into r' and thus q must additionally be increased by more than 1 because b is only doubled in each step.

Example for division - Case 3

Division for $a = 100$, $b = 80$ shall be calculated. The result for $100 / 160$ is known which is $q' = 0$ and $r' = 100$.

To reach the previous step, divisor of previous step 80 is compared to remainder r' : The constraint is violated as $100 \geq 80$!

This means that the divisor fits into the remainder, so it must be subtracted from the remainder and the quotient increased by 1.

Therefore $q = q' * 2 + 1 = 0 * 2 + 1 = 1$ and remainder $r = r' - b = 100 - 80 = 20$ which is correct as $100 / 80$ results in $q = 1$, remainder 20.

The same written in another notation:

```
100 = q * 80 + r.  
100 = q' * 160 + r'.  
-> 100 = 0 * 160 + 100. As 100 >= 80:  
100 = (2 * q' + 1) * 80 + (r' - b) = 1 * 80 + 20.
```

This can be implemented as follows:

```
void DivRem(SunBigUInt10 a, SunBigUInt10 b, out SunBigUInt10 q, out SunBigUInt10 r)  
{  
    if (a < b) /* Case 1 */  
    {  
        q = 0; r = a;  
    }  
    else  
    {  
        SunBigUInt10 qTmp, rTmp;  
        DivRem(a, 2 * b, out qTmp, out rTmp);  
        if (rTmp < b) /* Case 2 */  
        {  
            q = 2 * qTmp;  
            r = rTmp;  
        }  
        else /* Case 3 */  
        {  
            q = 2 * qTmp + 1;  
            r = rTmp - b;  
        }  
    }  
}
```

Note that there is a variant of the algorithm where instead of doubling divisor b , the dividend a is halved, see [4] for a nice explanation of this alternative.

[\[Back to top\]](#)

4.6 Exponentiation

Exponentiation of unsigned integers can be calculated extremely simple, as a^b is nothing else than multiply a with itself b -times: $a^b = a * a * \dots * a$. An iterative implementation using a for-loop is trivial, but has linear time complexity $O(n)$.

```
public static SunBigUInt10 PowSimple(SunBigUInt10 a, SunBigUInt10 b)  
{  
    SunBigUInt10 res = 1;  
    for (SunBigUInt10 i = 0; i < b; i++)  
    {  
        res = res * a;  
    }  
    return res;  
}
```

A slightly improved algorithm is the exponentiating by squaring method (see [3]) which is more efficient especially for large exponents b . It bases on following two observations:

- If b is even, then $a^b = (a^2)^{b/2}$.
- If b is odd, then $a^b = a * (a^2)^{(b-1)/2}$.

This results in following possible implementation:

```
public static SunBigUInt10 Pow(SunBigUInt10 a, SunBigUInt10 b)  
{  
    SunBigUInt10 res = 1;  
    while (b > 0)  
    {  
        if (b.IsOdd())  
        {  
            res *= a;  
        }  
        b /= 2;  
        a = a * a;  
    }  
    return res;  
}
```

4.7 Modular exponentiation (ModPow)

Modular exponentiation is an exponentiation performed over a modulus. It is defined as:

$c = v^e \bmod m$, where v is the value (sometimes also called base), e is the exponent and m the modulus.

This can be directly calculated (first v^e , then divide the result modulu), but the intermediate result of the exponentiation becomes quickly extremely large which makes this approach inefficient.

A slightly better algorithm is based on the fact that $(x * y) \bmod m = (x \bmod m * y \bmod m) \bmod m$.

Modular exponentiation - Example 1: $(12 * 23) \bmod 13$

Direct calculation: $(12 * 23) \bmod 13 = 276 \bmod 13 = 3$.

Stepwise calculation: $(12 * 23) \bmod 13 = (12 \bmod 13 * 23 \bmod 13) \bmod 13 = (12 * 10) \bmod 13 = 3$.

As $v^e \bmod m = (v * v \dots * v) \bmod m$ (with e -times the v factor) $= (v \bmod m * v \bmod m \dots * v \bmod m) \bmod m$, this can also be applied to calculate modular exponentiation by multiplying only the " $v \bmod m$ " factors.

Assuming $e = 3$, then:

$$\begin{aligned}
 & v^3 \bmod m \\
 &= (v \bmod m * v \bmod m * v \bmod m) \bmod m \\
 &= (c_0 * v \bmod m * v \bmod m) \bmod m \quad (\text{where } c_0 = v \bmod m) \\
 &= (c_1 * v \bmod m) \bmod m \quad (\text{where } c_1 = c_0 * v \bmod m) \\
 &= (c_1 * v) \bmod m \\
 &= c_2 \bmod m \quad (\text{where } c_2 = c_1 * v \bmod m) \\
 &= c
 \end{aligned}$$

Modular exponentiation - Example 2: $46^3 \bmod 17$

Direct calculation: $46^3 \bmod 17 = 97336 \bmod 17 = 11$

Stepwise calculation:

$$\begin{aligned}
 & 46^3 \bmod 17 \\
 &= (46 \bmod 17 * 46 \bmod 17 * 46 \bmod 17) \bmod 17 \\
 &= (12 * 46 \bmod 17 * 46 \bmod 17) \bmod 17 \quad \text{where } c_0 = 46 \bmod 17 = 12 \\
 &= (8 * 46 \bmod 17) \bmod 17 \quad \text{where } c_1 = 12 * 46 \bmod 17 = 8 \\
 &= (8 * 46) \bmod 17 \quad \text{where } c_2 = 8 * 46 \bmod 17 = 11 \\
 &= 11
 \end{aligned}$$

This can be implemented in an iterative way as follows:

```

public static SunBigUInt10 ModPow(SunBigUInt10 value, SunBigUInt10 exponent, SunBigUInt10
modulus)
{
    if (modulus == 1) return 0;
    SunBigUInt10 c = 1;
    for (SunBigUInt10 i = 0; i < exponent; i++)
    {
        c = (c * value).Mod(modulus);
    }
    return c;
}

```

4.8 Shifting

For the primitive integer types, which are stored binary which means with base = 2, a shift operation means the shifting of the bits of the number either to the left or to the right. A binary left shift of n bits is equal to the multiplication with 2^n and the right shift of n bits is equal to the division by 2^n .

As our big unsigned integer library uses base 10, the "natural" shifting is also adapted from base 10, i.e. single digits / array elements are shifted.

Shifting by base 10

The value 5831 is stored `byte[] digits = { 1, 3, 8, 5 }.`

Shifting left e.g. by 3 means now that `digits[0]` is moved to `digits[3]`, `digits[1]` to `digits[4]`, and so on.

The new digits at indices 0, 1 and 2 are filled with zeros. This results in `{ 0, 0, 0, 1, 3, 8, 5 }` which is 5831000 - in fact the original number is multiplied by $10^3 = 1000$.

Shifting right e.g. by 2 means now that `digits[2]` is moved to `digits[0]`, `digits[3]` to `digits[1]`, and so on.

There are digits shifted out which means precision has been lost. This results in `{ 8, 5 }` which is 85 - in fact the original number is divided by $10^2 = 100$.

So here a left shift of n bits is equal to the multiplication with 10^n and the right shift of n bits is equal to the division by 10^n .

The implementation of these shiftings are simple movements of array elements to new indices and adjusting the array length to the actual number of digits.

```
public static SunBigUInt10 LeftShiftByBase(SunBigUInt10 a, int b)
{
    if (b < 0) throw new ArgumentOutOfRangeException("b must not be negative");

    byte[] temp = new byte[a.digits.Length + b];
    for (int i = 0; i < a.digits.Length; i++)
    {
        temp[i + b] = a.digits[i];
    }
    return new SunBigUInt10(temp);
}

public static SunBigUInt10 RightShiftByBase(SunBigUInt10 a, int b)
{
    if (b >= a.digits.Length) return 0;

    byte[] temp = new byte[a.digits.Length - b];
    for (int i = a.digits.Length - 1; i >= b; i--)
    {
        temp[i - b] = a.digits[i];
    }
    return new SunBigUInt10(temp);
}
```

Of course, this might raise the question what to do with shifting by another value than base? For example, how about binary shifting?

Optimized shifting, that is implementing shifting by really shift the digits to left or right, can only be implemented for the used base.

For shifting using another base value, this has to be performed by explicit multiplication and division:

A binary left shift by n is equal to multiply a number x by 2^n : $x = x * \text{SunBigUInt10.Pow}(2, n)$.

Consecutively, a binary right shift by n is equal to divide a number x by 2^n : $x = x / \text{SunBigUInt10.Pow}(2, n)$.

[\[Back to top\]](#)

4.9 Comparison

Finally, it is necessary to compare two big unsigned integer objects to analyse their relation. Common operations are `==` (equality), `!=` (inequality), `>` (greater than), `>=` (greater or equal than), `<` (less than) and `<=` (less or equal than).

Equality (==):

The design of the library requires that the array holding the digits has exactly the number of relevant digits, i.e. there are no leading zeros. Two numbers cannot be equal if they have a different number of digits, so the length of the digits array of both numbers is a fast first check to exclude equality in case of different array lengths.

If both numbers have the same amount of digits, then each digit must be compared - if at least one digit is different, then both numbers are not equal.

There the implementation:

```
public static bool operator ==(SunBigUInt10 a, SunBigUInt10 b)
{
    if (a.digits.Length != b.digits.Length)
    {
        return false;
    }
    for (int i = a.digits.Length - 1; i >= 0; i--)
    {
        if (a.digits[i] != b.digits[i])
        {
            return false;
        }
    }
    return true;
}
```

Inequality (!=):

The simplest approach is to define inequality as the logical inverse of equality, thus $a \neq b \implies !(a == b)$.

```
public static bool operator !=(SunBigUInt10 a, SunBigUInt10 b)
{
    return !(a == b);
}
```

Greater than (>):

Based on the description of equality comparison, a number is greater than another one if it has more digits. The other way round, it is definitely smaller if it has less number of digits than the other number.

In case both numbers have the same amount of digits, then each digit must be compared, starting with the most significant digit. Once a single digit is greater than the digit of the other number, the whole number is greater. Following code snippet shows the implementation:

```
public static bool operator >(SunBigUInt10 a, SunBigUInt10 b)
{
    if (a.digits.Length > b.digits.Length)
    {
        return true;
    }

    if (a.digits.Length < b.digits.Length)
    {
        return false;
    }

    // a and b have same number of digits
    for (int i = a.digits.Length - 1; i >= 0; i--)
    {
        if (a.digits[i] > b.digits[i])
        {
            return true;
        }
        else if (a.digits[i] < b.digits[i])
        {
            return false;
        }
    }

    // both values are equal, return false
    return false;
}
```

Greater or equal than (>=):

As the name of this operator tells us, it must be checked if either the number is greater or if it is equal to another number. Both operators are already defined, so use them:

```
public static bool operator >=(SunBigUInt10 a, SunBigUInt10 b)
{
    return ((a > b) || (a == b));
}
```

Smaller than (<):

A number is smaller than another one, if it's *not* greater than or equal to this other number:

```
public static bool operator <(SunBigUInt10 a, SunBigUInt10 b)
{
    return !(a >= b);
}
```

Smaller or equal than (<=):

A number is smaller or equal than another one, if it's *not* greater than this other number:

```
public static bool operator <=(SunBigUInt10 a, SunBigUInt10 b)
{
    return !(a > b);
}
```

[\[Back to top\]](#)

4.10 Parity (Even / Odd)

A number is even if it's divisible by 2, otherwise it's odd. The way the numbers are stored makes this classification pretty simple, as a number in base 10 is even if it's least significant digit is even (0, 2, 4, 6 or 8).

```
public bool IsEven()
{
    return this.digits[0] % 2 == 0;
}

public bool IsOdd()
{
    return this.digits[0] % 2 != 0;
}
```

[\[Back to top\]](#)

5. Extending to other bases

The base 10 was chosen because according to my opinion, it's the easiest one to start dealing with arbitrary precision arithmetic. It feels natural as we are accustomed to use these numbers, and also debugging is easy. However, a byte is used to store one of the ten possible base 10 digits - while a byte can store 256 different values, so a huge amount of memory is wasted.

It is possible with only minor changes to use base 256 with the presented algorithms to use the memory more efficiently. Actually just replace '10' by '256' and there you go. Only the semantics of the shift algorithms change as the one digit shift means the multiplication / division by 256 instead of 10. See the linked source code at the top of this page for an implementation with base 256.

Of course it is also possible to use primitives like short, int and long and further increase the radix of the numeral system. But be ware that several methods use arithmetic operations like addition and multiplication of these primitive types. For example, the presented multiplication algorithm multiplies two digits and adds one carry digit. With base 10, the maximum value is $9 * 9 + 9 = 90$ that fits in an integer. In general with base b , the maximum value is $(b-1) * (b-1) + (b-1) = b^2 - b$ which must be taken into account to not overflow.

[\[Back to top\]](#)

6. Conclusion & References

The presented approach is intentionally kept very simple to provide a starting point into the world of arbitrary precision arithmetic. In case of further interest into this topic, I suggest to investigate the linked resourcea and just search the internet for "big integer library source code" and "arbitrary precision arithmetic" to gather more sophisticated information and to analyze existing big integer libraries. The used data storage and algorithms are by far more efficient, but also harder to understand.

That's it! Hope it was interesting for you and have fun and learned something! Keep coding!

Sunshine, August 2020

References

- [1] [Arbitrary precision arithmetic @ Wikipedia](#)
- [2] [Exponentiation by squaring @ Wikipedia](#)
- [3] [Modular exponentiation @ Wikipedia](#)
- [4] [Recursive division algorithm for two n bit numbers @Stackoverflow](#)
- [5] Knuth, 1981. The Art of Computer Programming: Volume 2, Seminumerical Algorithms
- [6] Hanson, 1996. C Interfaces and Implementations: Techniques for Creating Reusable Software

History

- 2020/08/19: Initial version.

This site is part of [Sunshine's Homepage](#)