Fast conversion from a String to an arbitrary precision number.

By Henrik Vestermark (hve@hvks.com)

Abstract

This is a follow-up to a previous paper that describes a very fast method to convert an arbitrary precision number to a string. This paper describes the opposite process of converting a string number to an arbitrary precision number. We all know how to convert a string number to simple language types like integer and float using basic encoding technics. However, how does it stack up when the string contains 1,000 to 1M string digits or even more when converting it to an arbitrary precision number?
This paper describes the issue with the basic conversion algorithm and outlines a new improved algorithm that speeds up the process by a factor of more than 20,000.

Introduction

This issue come to light when I was finished writing a new version of my Arbitrary Precision Math library where I converted the internal format from a decimal base string form (where each arbitrary precision number was stored as a decimal string) to a binary form (where the arbitrary precision number was stored as a vector of binary digits). In the conversion process everything when smoothly until I began to test the performance of the library. The internal arithmetic handling was fast for all the usual types like +,-,*,/ etc. as expected when using binary digits instead of decimal digits. However, when I was testing the initialization of an arbitrary precision number with a string, the performance when 'south' when handling more than a few thousand digits. To improve the performance I outline here a new way to handle this conversion for string digits of a million digits or more.

Change log

2-March-2023. Minor corrections and cleaning up the document.

Contents

Abstract: 1
Introduction: 1
A simple approach for conversion from string form 3
The Arbitrary precision library 3
1st Approach 4
2nd Approach 5
3rd Approach 6
String numbers in other bases 9
Arbitrary precision numbers 9
Conclusion 10
Reference 10

A simple approach for conversion from string form

As taught in computer language classes, we learn how to convert a decimal string to binary numbers and there are several possibilities to accomplish this.

1) Use the library sscanf() function.
2) Use atoi(), strtol(), or similar if you were dealing with integers.
3) Use cin >> to a variable;
4) Write your little code block.
5) Other methods.

To make it simple we will only discuss the method for converting a string to an arbitrary precision integer, although with few changes it can be adapted to handle conversion from a decimal floating-point string to an arbitrary precision floating point. We will initially use a common string as the input to our arbitrary precision number, based on the STL library string class. We will also briefly discuss when the input string is not in a decimal form but represent other bases like hexadecimal, octal, or binary string numbers. Usually, the input process does one decimal digit conversion at a time as used in the following pseudo algorithm. The algorithm is based on building the integer number one digit at a time.

The algorithm is as follows:

1. Take the first decimal string digit and convert it to a binary digit
2. Multiply the sum by the base (10)
3. Add the single binary digit from 1 to the sum
4. Repeat the process until all digits have been converted

The function standard() can easily be written, see below:

```
uintmax_t standard(const string& s)
	{
	size_ t i;
	uintmax_t number = 0;
	for (i = 0; i < s.size(); ++i)
		{
		number *= 10;
		number += s[i] - '0';
		}
	return number;
	}
```

uintmax_t is the biggest integer available. In most systems, it is a 64-bit quantity. Overall, basic stuff.

The Arbitrary precision library

The int_precision class is a signed arbitrary precision variable. It has two members in the class.

1) int mSign
2) vector<uintmax_t> mBinary

The mSign holds the sign of the int_precision number and can be either +1 for positive numbers or -1 for negative numbers.
The mBinary is a vector of uintmax_t binary numbers (64bit unsigned integer)

The first entry is the least significant 64-bit of the arbitrary number and the increased vector index is increasing the significance of the number. The highest entry at mBinary.size()-1 is the most significant 64-bit of the arbitrary precision number.

For the algorithm presented here, you do not need to know any details of the internal of the int_precision class.

1st Approach

Using the author's arbitrary precision library we can now make our first approach to handle an unlimited number of decimals in a string for conversion to arbitrary precision. The arbitrary version of the C++ type, int is the type int_precision. Our first approach is simply to replace the declaration with the uintmax_t with the int_precision declaration to create our first solution.

```
int_precision firstapproach(const string& s)
       {
       size_t i;
       int_precision number = 0;
       for (i = 0; i < s.size(); ++i)
              {
              number *= 10;
              number += s[i] - '0';
              }
       return number;
       }
```

Now we can throw strings containing thousands, millions, or even higher string numbers and it will all be converted to the internal binary version of the string.

Testing the solutions from 100 decimal digits up to 1M decimal string digits, you get the following performance.

String to Binary conversion	Time in msec
100	0.07
1,000	2.63
10,000	245
100,000	26,648

1,000,000	2,626,000 (2,636 sec)

Up to around 10,000 decimal digits, you get an acceptable performance but then it goes "south" from here, and for 1M digits number the conversion takes 2,626 Seconds, or more than 40 minutes, which of course is not acceptable.

2nd Approach

Our next approach is to recognize that the multiplication for every digit is where the vast majority of the time is spent and performance gets worse the more digits you are dealing with. Instead of doing one digit at a time you can do 19 digits at a time using the C library strtoull() functions. That way we use a "native" calculation to process up to 19 digits at a time and then only do the more expensive arbitrary precision multiplication for every 19 digits. This is not a new thing but something you will see in countless other implementations of an arbitrary precision package.

The algorithm for our 2nd approach is below.

```
int_precision secondapproach(const string& s)
	{
	int_precision number = 0;
	size_t i, length = s.size();
	const size_t max_digits = 19;
	uintmax_t pwr = _powerof10Table[max_digits];

	for (i = 0; length >= max_digits; length -= max_digits, i += max_digits)
		{
		number *= pwr;
		number += strtoull(s.substr(i, max_digits).c_str(), NULL, BASE_10);
		}

	if (length != 0)
		{
		number *= _powerof10Table[length];
		number += strtoull(s.substr(i, length).c_str(), NULL, BASE_10);
		}

	return number;
	}
```

The constant table _powerof10Table holds the power for each digit from 0 to 19 and is not shown here. With this approach, we limit the calculation of both the multiplication and the addition for arbitrary precision to once every 19 digits to process.

The performance is of course a lot better as indicated in below performance table

String to Binary conversion	Time in msec	
	1st approach	2nd approach

100	0.07	0.008
1,000	2.63	0.16
10,000	245	12.6
100,000	26,648	1,315
1,000,000	2,626,000 (2636 sec)	134,601 (135 sec)

Not surprisingly, our 2nd approach is much faster and gives nearly acceptable performance however, as you can see 100k digits are done in approx. 1.3sec, but 1M digit is ~ 135sec. The worst-case behavior is still ~n^2 which make this approach not suitable for number exceeding 100-200k digits.

Can we do better than this? The answer is yes, as developed in our 3rd approach.

3rd Approach

The problem with our two first algorithms is that the multiplication is performed using an increasing number of digits slowing the process down as the number of digits gets higher. The n^2 behavior is not suitable for a very large number of decimal digits.

In our 3rd approach, we divide the new algorithm into 2 steps.

The first step is just to process the decimal string into trunks of 19 digits using the c library built-in function strtoull() (64bit) and save them in a vector (array) of int_precision numbers to be used in step2 of the new algorithm.

The vector is declared as:

```
        vector<int_precision> vn(0);
```

With a size of zero (no elements). We then process the initial string backward 19 digits at a time and stored each 19-digit trunk in the vector. Since we push it from the back onto the vector, we get trunks of 19 digits with increasing significance. The variable length is the size of our string, *s*, that holds the decimal string. max_digits is the constant 19 representing the maximum number a 64-bit unsigned integer can hold.

```
// Step 1 partition the string into a (single size) int_precision vector in order from most to
least significant
for (; length > max_digits; length -= max_digits)
        vn.push_back(strtoull(s.substr(length - max_digits, max_digits).c_str(), NULL, BASE_10));
vn.push_back(strtoull(s.substr(0, length).c_str(), NULL, BASE_10));
```

The illustration below shows how a decimal string is processed and stored as an element in the vector of int_precision numbers.

String: "1234567'1234567890123456789'1234567890123456789'1234567890123456789'1234567890123456789'1234567890123456789"

0	1234567890123456789
1	1234567890123456789
2	1234567890123456789
3	1234567890123456789
4	1234567890123456789
6	1234567

Algorithm

- Starting backwards with the last 19 digits
- Continue pushing 19 digits at a time to the vector
- Last push contain 1..19 digits

Now the real-time saving or performance gain comes from the processing of the initial vector into the final number. We do that by repeatedly doing a pairwise calculation of two adjacent numbers multiplied by the radix. Since the number is initially stored as 19-digit entries, the initial radix is 10^{19}. The first loop of running through the vector will create a new vector with half the entries rounded to an even number of vector elements. We reuse the existing vector instead of storing it in a new vector. Since we have 6 elements in the first loop will create a new vector with 3 elements.

String: "1234567'1234567890123456789'1234567890123456789'1234567890123456789'1234567890123456789'1234567890123456789"

1st Iteration

- Recalculate pair wise
- Radix is 10^{19}
- Stored in the same vector
- Resize vector to 3 half the size of starting vector

0	1234567890123456789
1	1234567890123456789
2	1234567890123456789
3	1234567890123456789
4	1234567890123456789
6	1234567

V[0]=V[0]+V[1]*radix
V[1]=V[2]+V[3]*radix →
V[2]=V[4]+V[5]*radix

0	12345678901234567891234567890123456789
1	12345678901234567891234567890123456789
2	12345671234567890123456789

The first new entry is created by: v[0]=V[0]+V[1]*radix etc.
Moreover, we continuously loop until the vector size is down to one. For each loop, we square the radix

String: "1234567'1234567890123456789'1234567890123456789'1234567890123456789'1234567890123456789'1234567890123456789"

Vector of Int_precision's

2nd Iteration

- Recalculate pair wise
- Radix is squared to 10^{38}
- Stored in the same vector
- Resize vector to 2

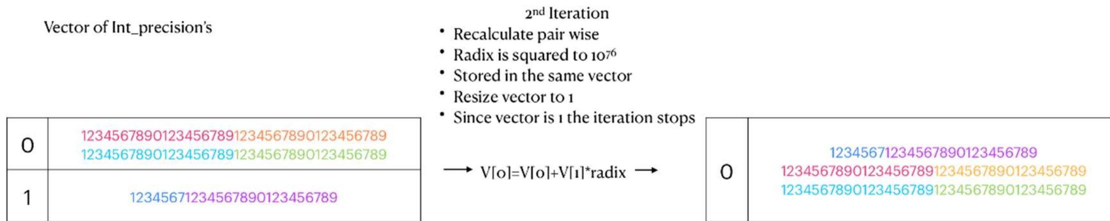
0	12345678901234567891234567890123456789
1	12345678901234567891234567890123456789
2	12345671234567890123456789

→ V[0]=V[0]+V[1]*radix →
V[1]=V[2]

0	12345678901234567891234567890123456789 12345678901234567891234567890123456789
1	12345671234567890123456789

And the last loop

String: "1234567'12345678901234567890'12345678901234567890'12345678901234567890'12345678901234567890'12345678901234567890"



Now the first entry v[0] contains the binary counterpart to the decimal string.
The algorithm reduces the number of multiplications with a high number of digits and creates an algorithm that scales much better than the two first algorithms.

The final algorithm is shown below and is surprisingly simple:

```
int_precision thirdapproach(const std::string& s)
	{
	const size_t max_digits = 19;
	size_t length =s.size(), i, j;
	vector<int_precision> vn(0);
	int_precision radix=_powerof10Table[max_digits];
	// Reserve enough space to avoid resizing of vector
	vn.reserve( length / max_digits + 16);
	// Step 1 partition the string into a (single size) int_precision vector
	// in order from most to least significant
	for (; length > max_digits; length -= max_digits)
		vn.push_back(strtoull(s.substr(length - max_digits,
max_digits).c_str(), NULL, BASE_10));
	vn.push_back(strtoull(s.substr(0, length).c_str(), NULL, BASE_10));

	// Step2 collected into higher binary values by reducing the vector
	// to half its size per iteration
	while( vn.size() > 1 )
		{
		for (i = 0, j = 0; j < vn.size(); ++i, j += 2)
			{
			if (j + 1 < vn.size())
				vn[i] = vn[j] + vn[j + 1] * radix;
			else
				vn[i] = vn[j];
			}
		vn.resize(i); // Resize the vector to half it size
		if (i > 1)
			radix *= radix;  // Update the radix
		}

	return vn[0];
	}
```

The performance is also considerably better as shown in this table

String to Binary conversion	Time in msec		
	1st approach	2nd approach	3rd approach
100	0.07	0.008	0.008
1,000	2.63	0.16	0.10
10,000	245	12.6	4.9
100,000	26,648	1,315	147
1,000,000	2,626,000 (2636 sec)	134,601	1,371
10,000,000	-	13,888,000	18,767

With the new algorithm, we can now process 1M decimal digits in less than 1,4 seconds and only 18.7 sec for 10M digits. Based on the above table it is very clear that the 2nd approach scales poorly (O(n^2) while the 3rd approach scale linearly and therefore has a much improved worst-case behavior compared to the 2nd approach.

String numbers in other bases

Until now, we have only dealt with decimal string numbers. However using standard C++ notation you could call the function with string numbers in Hexadecimal (base 16), Octal (base 8), and binary (base 2).

For base 2 and base 16, we can use a shortcut. The internal of the int_precision is stored in vectors of uintmax_t (unsigned 64-bit entries) and we can therefore directly map it into the int_precision vector stored as mBinary vector in the class definition. A hexadecimal number is exactly 4-bits wide and we can therefore store 16 hexadecimal digits into one internal vector entry in mBinary. This is essentially step 1 of the new algorithm avoiding step two entirely. The same goes for base two where we could store exactly 64 base 2 digits into one internal vector entry in mBinary and again avoid the step two process. With Octal, we are not so lucky. In addition, an octal number requires 3 bits, therefore 21 octal digits will require 63 bits, and 22 octal digits will require 66 bits. We will need to do the same as we did for the decimal handling when it does not divide evenly up into 64 bits.

Here we can group the octal number into trunks of 21 digits at a time in step one of the algorithms. Set the initial radix to 8^{21} and then perform step two as we did for the decimal algorithm.

Arbitrary precision numbers

Until now we have avoided any discussion about floating point numbers. However, we can use the same algorithm as outlined here. A floating-point number has the format

xxxx.yyyyyyyEeeee

Where xxxx is the digit in front of the period '.'
yyyyyyy is the fraction part.
E is the exponent designator and eeee is the signed exponent.

If we ignore the fraction character '.' then xxxyyyyyy is an integer number and we can use the algorithm layout in this document. After the calculation, we can adjust for the number of fraction digits by multiplying the result with $10^{-numberoffractiondigits+eeee}$ to get the correct result.

Conclusion

The above new methods show that you can gain a significant performance improvement by re-defining the algorithm using pairwise calculation with radix squaring.

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages (hvks.com)