Fast Computation of Stirling Numbers in Arbitrary Precision.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This is a continuation of the series of papers written dealing with the practical aspect of implementing an arbitrary precision math package. The paper describes how to efficiently generate the Stirling number of the first, second, and third kind (the third kind is also known as the Lah number), needed for making a complete Arbitrary precision Math package.

Introduction:

We will look at Stirling's number of the first, second, and third kinds. The latter is also known as the Lah number

As usual, we will show the actual C++ source for the calculation using the author's own arbitrary precision Math library, see [1].

This paper is part of a series of arbitrary precision papers describing methods, implementation details, and optimization techniques. These papers can be found on my website at www.hvks.com/Numerical/papers.html and are listed below:

1. Fast Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Square Root & Inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
4. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision
5. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision
6. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
7. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms
8. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision
9. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
10. Fast conversion from arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string
11. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision

Contents

The Arbitrary precision library

If you already are familiar with the arbitrary precision library, you can skip this section. There are two classes. One for *int_precision* that handle arbitrary precision integers and one for *float_precision* that handles all *floating-point* arbitrary precision. Since Stirling numbers are integers we only need to highlight the int_precision class.

Int_precision class

To understand the C++ code and text we have to highlight a few features of the arbitrary precision library where the class name is *int_precision*. Instead of declaring, a variable with any of the build-in integer type char, short, int, long, long long, unsigned char, unsigned short, unsigned int, unsigned long, and unsigned long long you just replace the type name with *int_precision*. E.g.

```
int_precision ip;  // Declare an arbitrary precision integer
```

You can do any integer operations with *int_precision* that you can do for any type of integer in C++. Furthermore, there are a few methods you will need to know.
One of them is .iszero() which simply returns true or false if the *int_precision* variable is zero or not zero. Another is .even() and .odd() which return the Boolean value of the number even and odd status. There are other methods but I will refer you to the user manual for the arbitrary precision package [1].

Internal format for int_precision variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mNumber;
```

uintmax_t is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integer to store our integer precision number.

The method .size() returns the number of internal vector entries needed to hold the number.

The number is stored such that the vector mNumber[0] holds the least significant 64-bit binary data. The mNumber[size()-1] holds the most significant 64-bit binary data. The sign is kept separately in a class field variable mSign, which means that the mNumber holds the unsigned binary vector data.

For more details see [1].

Stirling Number

Stirling numbers were invented around 1730 by J. Stirling. There are today Stirling numbers of the first kid, Stirling numbers of the second kind, and then Lah numbers (sometimes referred to as Stirling numbers of the third kind). To be true to the name of this paper we will refer to Lah numbers as Stirling numbers of the third kind.
For all three kinds, they describe the coefficients relating to three different sequences of polynomials that are common in combinatorics. Moreover, all three can be defined as the number of partitions of n elements into k non-empty subsets, with different ways of counting orderings within each subset. However, as described below there are several different applications for the Stirling number.
For the Stirling number of the first kind they typically use the notation s(n,k). For the Stirling number of the second kind, they use the notation S(n,k). And finally, for Lah (the Stirling number of the third kind) we use the notation L(n,k).

Application for Stirling number [7]

The Stirling numbers of the first, second, and the third kind are mathematical concepts that have several applications in various fields, including:

- Combinatorics: Stirling numbers are used in combinatorial problems that involve counting the number of ways to partition a set into non-empty subsets or to arrange objects into cycles or permutations. For example, the Stirling numbers of the second kind count the number of ways to partition a set of n objects into k non-empty subsets, while the Stirling numbers of the first kind count the number of permutations of n objects with k disjoint cycles.
- Probability: Stirling numbers are used in the study of probability distributions, such as the Poisson distribution and the binomial distribution. They can be used to compute the probability of a certain number of events occurring in a given time or space.
- Statistics: Stirling numbers are used in statistics to analyze data and estimate parameters of statistical models. For example, the Stirling numbers of the second kind can be used to estimate the number of distinct groups or clusters in a dataset, while the Stirling numbers of the third kind can be used to estimate the number of ways to distribute n objects into k cells.
- Number theory: Stirling numbers are used in number theory to study the properties of integers and their partitions. They can be used to compute the number of partitions of an integer into k parts or parts of a certain size.
- Physics: Stirling numbers are used in quantum mechanics to calculate the matrix elements of certain operators, such as the creation and annihilation operators, that describe the behavior of particles in a quantum system.

Overall, the Stirling numbers have a wide range of applications in mathematics and its various applications, and their properties and relationships with other mathematical concepts make them a valuable tool for solving problems in various fields.

Stirling Number of the first kind

The notation for the signed Stirling numbers of the first kind is s(n,k) and the definition of the Stirling numbers of the first kind was that they are the coefficients s(n,k) of the falling factorial expanded:

$$x^{\underline{n}}=x(x-1)(x-2)\ldots(x-n+1) \qquad (1)$$

Into powers of x:

$$x^{\underline{n}} = \sum_{k=0}^{n} s(n,k)x^k = s(n,0)x^0 + s(n,1)x + s(n,2)x^2 \ldots s(n,n)x^n \qquad (2)$$

This is the signed Stirling number of the first kind. Similarly, we also have the rising factorial defined as:

$$x^{\underline{n}}=x(x+1)(x+2)\ldots(x+n-1) \qquad (3)$$

is a polynomial of x with degree n, whose expansion is:

$$x^{\overline{n}} = \sum_{k=0}^{n} c(n,k)x^k = \sum_{k=0}^{n} \begin{bmatrix} n \\ k \end{bmatrix} x^k \qquad (4)$$

This is the unsigned Stirling number of the first kind. The unsigned Stirling number of the first kind is usually denoted with c(n,k) or just $\begin{bmatrix} n \\ k \end{bmatrix}$.

The unsigned number of the first kind can be calculated as a recurrence using the relation:

$$\begin{bmatrix} n+1 \\ k \end{bmatrix} = n \begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix} \; for\ k > 0 \qquad (5)$$

We have some initial conditions:

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1,\ \begin{bmatrix} n \\ 0 \end{bmatrix} = 0,\ \begin{bmatrix} 0 \\ n \end{bmatrix} = 0 \qquad (6)$$

Algorithm 1 Recurrence for the unsigned Stirling number of the first kind.

This is the same as writing c(n+1,k)=n· c(n,k)+c(n,k-1).

And for the signed number of the first kind, we can use a similar recurrence relation:

$$\begin{bmatrix} n+1 \\ k \end{bmatrix} = -n \begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix} \; for\ k > 0 \qquad (7)$$

With the same initial conditions as for the unsigned version.

Algorithm 2 recurrences for the signed Stirling number of the first kind.

Same as writing s(n+1,k)=-n· s(n,k)+s(n,k-1).

The relationship between the signed and unsigned Stirling number of the first kind is:

$$s(n,k) = (-1)^{n-k} c(n,k) = (-1)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} \tag{8}$$

Two other simple identities can come in handy when needed.

$$\boxed{\begin{bmatrix} n \\ n \end{bmatrix} = 1, \begin{bmatrix} n \\ 1 \end{bmatrix} = (n-1)! \qquad (9)}$$

This recurrence relation is pretty simple to implement in the C++ language where we can combine the unsigned and signed version of the Stirling number of the first kind into this recursive function. In the below function, we use the two above algorithms one and two where only the sign differs.

Source Stirling number of the first kind. Recursion
```
static intmax_t stirling_first(int n, int k, const bool sign =false )
{
	if (k==0 && n==0 || k==n) return 1;
	if (k==0 || n==0 || k>n) return 0;
	return stirling_first(n-1,k-1,sign)+(sign?-(n-1):+(n-1))*
		stirling_first(n-1, k, sign);
}
```

We notice that we call the function stirling_first twice per call and the total number of recursion calls can be calculated as $2\binom{n+1}{k} - 1$. Where $\binom{n+1}{k}$ is the binomial coefficient. E.g. for n=20 and k=10, you get 705,431recursive calls. For n=100 and k=50 you get 399,608,854,866,744,452,032,002,440,111. This is of course not manageable from a practical point of view. We will have to do something to improve the performance of the formula.

The trick is as usual to change from recursion to looping and add some memory-saving construction. I usually start with saving the temporary loop data in an array of size (n+1,k+1) representing the various values of c(n,k). From there I realized that as we progress through the matrix we only need access to the previous matrix row building up the current matrix row. In other words, we need to hold two vectors, the current row vector and the previous row vector as outlined below.

Source Stirling number of the first kind. Looping for arbitrary precision
```
static int_precision stirling_first(int_precision& nip, int_precision& kip, const
bool sign = false)
{
	const int_precision c0(0), c1(1);
	const int n = int(nip), k = int(kip);
	std::vector<int_precision> prev((size_t)k + 1), current((size_t)k + 1);
	int_precision im1;

	if (k==0 && n==0 || k==n) return c1;
	if (k==0 || n==0 || k>n) return c0;
	prev[0] = c1; current[0] = c0;
	for (int i = 1; i <= n; ++i)
	{
		im1 = int_precision(i - 1);
		for (int j = 1; j <= k; ++j)
			current[j] = prev[j - 1] + im1 * prev[j];
		// Copy current to prev except in the last iteration where i==n
		if(i<n)
			prev = current;
	}
	// Return result of s(n,k) or c(n,k)
	return sign==true && (n-k)&0x1 ? -current[k] : current[k];
}
```

We have to replace the recursion with two double loops and two vectors that keep track of the progress. The outer loop loops from 1…n, and the inner loop from 1…k. The current vector is the vector building up c(n,k) and the vector prev is the previous loop or c(n-1,k).

In [6] they also list a very efficient algorithm that further reduces the need for vectors to a single vector by letting the second loop, looping backward and we end up with our final version optimized version below.

Source Stirling number of the first kind. Optimized looping.

```
static int_precision stirling_first_optimizedloop(int_precision& nip,
int_precision& kip, const bool sign = false)
{
	const int_precision c0(0), c1(1);
	const int n = int(nip), k = int(kip);
	std::vector<int_precision> cnk((size_t)n + 1,0);
	int_precision im1;

	if (k==0 && n==0 || k==n) return c1;
	if (k==0 || n==0 || k>n) return c0;

	cnk[1] = c1;
	for (int i = 1; i < n; ++i)
	{
		im1 = int_precision(i);
		// Do it from the back to the front of the vector
		for (int j = i+1; j > 0; --j)
			cnk[j] = cnk[j - 1] + im1 * cnk[j];
	}
	// Return result of s(n,k) or c(n,k)
	return sign == true && (n - k) & 0x1 ? -cnk[k] : cnk[k];
}
```

Performance of Stirling's first kind.

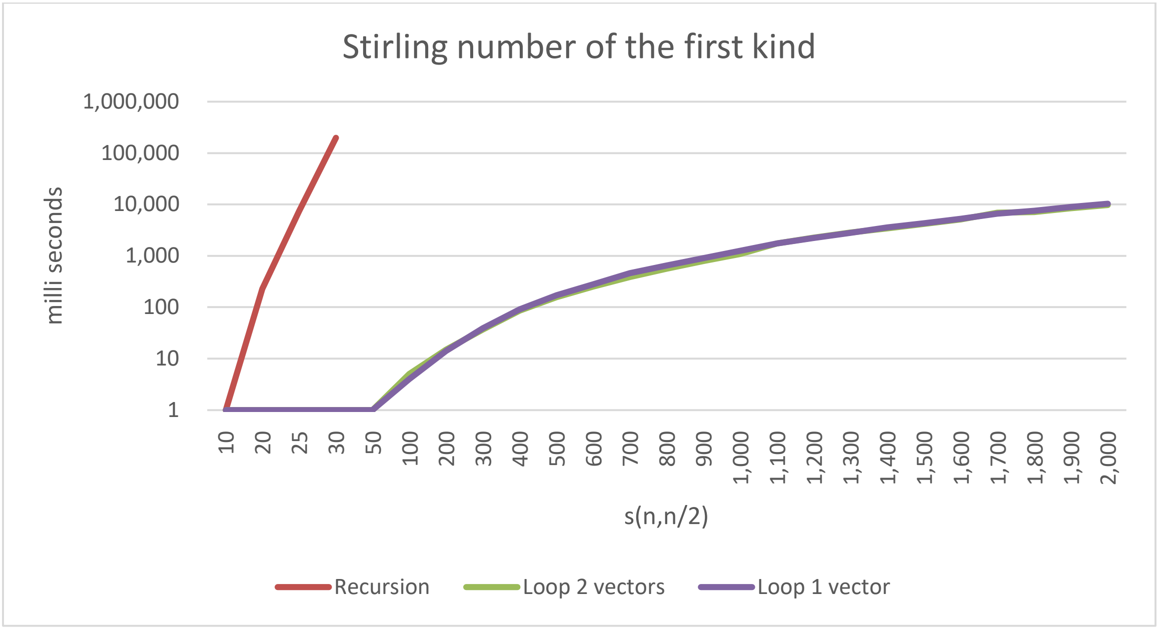

Figure 1

As indicated in the performance char above of the Stirling of the first kind. the recursion version is many magnitudes slower than the loop-based versions, which is expected, however, the surprise here is how much slower it is.

Recommendation for the Stirling number of the first kind

As the performance chart indicates above I recommend using one of the loop-based versions.

Stirling Number of the second kind

The Stirling number of the second kind is denoted as S(n,k) or $\left\{ {n \atop k} \right\}$ and is defined by the direct formula:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!}\sum_{i=0}^{k}(-1)^i \binom{k}{i} (k-i)^n \qquad (10)$$

It also obeys the following recurrence.

$$\left\{ {n+1 \atop k} \right\} = k\left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\} \ for\ k > 0\ and\ k < n \qquad (11)$$

We have some initial conditions:

$$\begin{Bmatrix} n \\ n \end{Bmatrix} = 1 \; for\; n \geq 0, \; \begin{Bmatrix} n \\ 0 \end{Bmatrix} = 0, \; \begin{Bmatrix} 0 \\ n \end{Bmatrix} = 0, for\; n > 0 \tag{12}$$

Algorithm 3. Stirling number of the second kind.

This is very similar to the Stirling number of the first kind, except that we multiply with k instead of n.

The recurrence can also be expressed like: S(n+1,k)=k· S(n,k)+S(n,k-1).

Not surprisingly, we end up with nearly the same recurrence function.

Source Stirling number of the second kind. Recursion for arbitrary precision

```
static int_precision stirling_second(int_precision& n, int_precision& k)
{
	const int_precision c0(0), c1(1);

	if (k.iszero() && n.iszero() || k == n) return c1;
	if (k.iszero() || n.iszero() || k > n) return c0;
	return stirling_second(n-c1,k-c1) + k * stirling_second(n-c1, k);
}
```

As we saw in Stirling number of the first kind we call the function stirling_second twice per call and the total number of recursion calls can be calculated as $2\binom{n+1}{k} - 1$. Where $\binom{n+1}{k}$ is the binomial coefficient. E.g. for n=20 and k=10, you get 705,431recursive calls. For n=100 and k=50 you get 399,608,854,866,744,452,032,002,440,111. We reached the same conclusion as for the Stirling number of the first kind that this is not manageable from a practical point of view. Converting the function to a loop-based you get a several magnitudes increase in performance.

Source Stirling number of the second kind. Optimized looping.

```
static int_precision stirling_second_loop(int_precision& nip, int_precision& kip)
	{
	const int_precision c0(0), c1(1);
	const int n = int(nip), k = int(kip);

	if (k==0 && n==0 || k==n) return c1;
	if (k==0 || n==0 || k>n) return c0;

	const int size = n - k;
	vector<int_precision> S(size + 1, c1);

	for (int i = 2; i <= k; ++i)
	{
		for (int j = 1; j <= size; ++j)
			S[j] += int_precision(i) * S[j-1];
	}
	return S[size];
}
```

Source. Stirling number of the second kind. Direct formula

```
static int_precision stirling_second_direct(int_precision& nip, int_precision&
kip)
{
	const int_precision c0(0), c1(1);
	const int n = int(nip), k = int(kip);
	int_precision sum(0), ipi(0);

	if (k == 0 && n == 0 || k == n) return c1;
	if (k == 0 || n == 0 || k > n) return c0;

	for (int i = 0; i <= k; ++i, ++ipi)
	{
		sum += (i&0x1 ? -c1 : +c1) * binomial(kip, ipi)*ipow(kip-ipi, nip);
	}
	sum /= factorial(kip);
	return sum;
}
```

Performance of Stirling's second kind.

As can be viewed below graph using recursion to find the Stirling number of the second kind is several magnitudes slower that the more direct methods, like looping or using the direct formula.

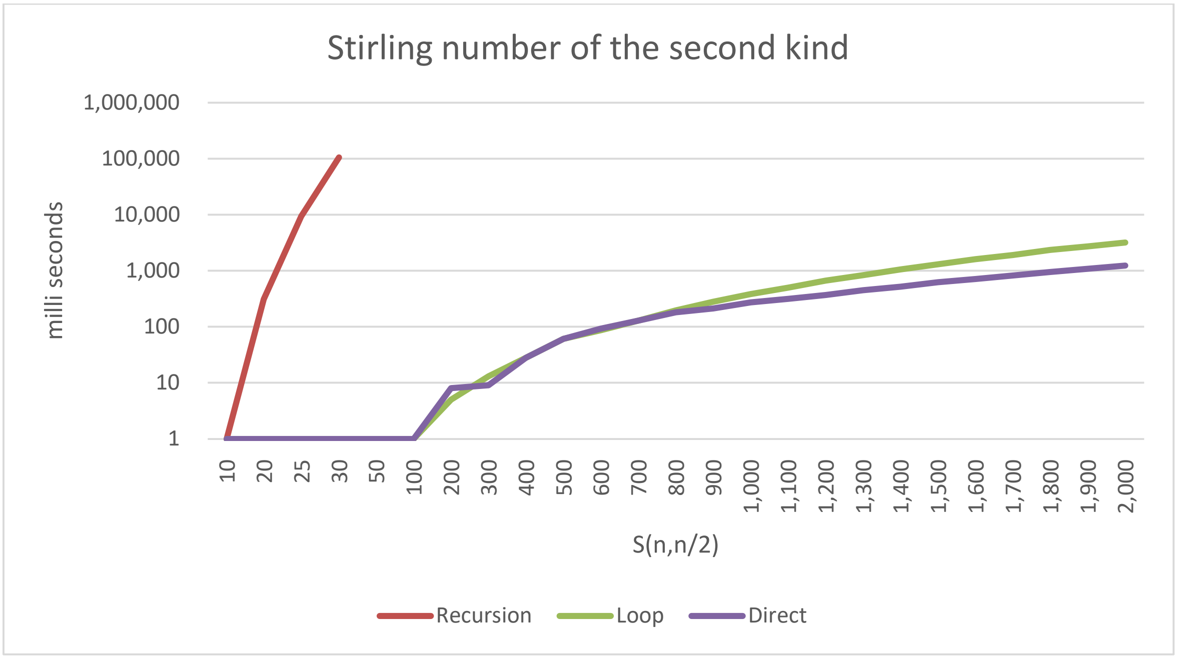

Figure 2. Performance of the Stirling number of the second kind.

Recommendation for the Stirling number of the second kind

As the performance is similar to the first kind figure 1. I recommend using the direct formula instead of either the recursion or loop-based versions.

Stirling Number of the third kind

Is better known as Lah numbers, which were discovered by I. Lah in 1954 and are coefficients that connect rising and falling factorials. Like the Stirling number of the first kind, the Stirling number of the third kind has both an unsigned and signed sequence. The unsigned Stirling number of the third kind is denoted by L(n,k) and is defined as:

$$L(n,k)=\binom{n-1}{k-1}\frac{n!}{k!} \tag{13}$$

A Signed Stirling number of the third kind is denoted by L'(n,k) and is defined as:

$$L'(n,k)=(-1)^n\binom{n-1}{k-1}\frac{n!}{k!} \tag{14}$$

The Stirling number of the third kind can also be expressed as a sum of the Stirling number of the first and second kind by:

$$L(n,k)=\sum_{j=0}^{n}\begin{bmatrix}n\\j\end{bmatrix}\begin{Bmatrix}j\\k\end{Bmatrix} \tag{15}$$

Where [] is the notation of the Stirling number of the first kind and {} is the notation for the Stirling number of the second kind.
The way the Stirling number of the third kind connects the rising and falling factorial is through the equations:

$$x^{\overline{n}}=\sum_{k=1}^{n}L(n,k)\cdot x^{\underline{k}} \tag{16}$$

And

$$x^{\underline{n}}=\sum_{k=1}^{n}(-1)^{n-k}L(n,k)\cdot x^{\overline{k}} \tag{17}$$

The unsigned Stirling number of the third kind can also be calculated as a recurrence using the relation:

$$L(n,k)=(n+k-1)L(n-1,k)+L(n-1,k-1) \tag{18}$$

We have some initial conditions:

$$L(n,0)=L(0,k)=0, L(n,k)=0\ for\ k>n\ and\ L(n,n)=1 \tag{19}$$

Algorithm 4. Unsigned Stirling number of the third kind

Another identity that can speed up the calculation is:

$$L(n,1)=n! \quad (20)$$

For the signed Stirling number of the third kind we have the following recurrence.

$$L'(n,k)=-(n+k-1)L(n-1,k)-L(n-1,k-1) \quad (21)$$

We have some initial conditions:

$$L'(n,0)=L'(0,k)=0, L'(n,k)=0\ for\ k>n\ and\ L'(n,n)=(-1)^n \quad (22)$$

Algorithm 5. Signed Stirling number of the third kind.

If we also want to use the (20) then we need to make it “signed”.

$$L'(n,1)=(-1)^n n! \quad (23)$$

Source code for the Stirling number of the third kind using recurrence (18 or 21).

```
static int_precision stirling_third(int_precision& n, int_precision& k, const
bool sign = false)
{
	const int_precision c0(0), c1(1);
	int_precision res;

	if (n == k) return (sign && n.odd()? -c1: +c1);
	if (k==0||n==0||k>n) return c0;
	if (k==1) return (sign && n.odd() ? -c1: c1)*factorial(n);

	res = (n+k-c1)*stirling_third(n-c1,k,sign) +
		stirling_third(n-c1,k-c1,sign);
	if(sign) return -res;
	return res;
}
```

Source code for the Stirling number of the third kind using first and second kind (15)

```
static int_precision stirling_thirdhybrid(int_precision& n, int_precision& k,
const bool sign = false)
{
	const int_precision c1(1);
	int_precision sum(0);

	for (int j = 0; j <= n; ++j)
		sum += stirling_first(n, int_precision(j),false) *
			stirling_second(int_precision(j), k);
	if (sign && n.odd())
		sum.change_sign();
	return sum;
}
```

Source code for the Stirling number of the third kind using (13) and (14)

```
static int_precision stirling_thirddirect(int_precision& n, int_precision& k,
const bool sign = false)
{
	const int_precision c1(1);
	int_precision res;

	res = binomial(n-c1, k-c1)*fallingfactorial(k+c1, n);
	if (sign && n.odd())
		res.change_sign();
	return res;
}
```

Performance of Stirling's third kind.

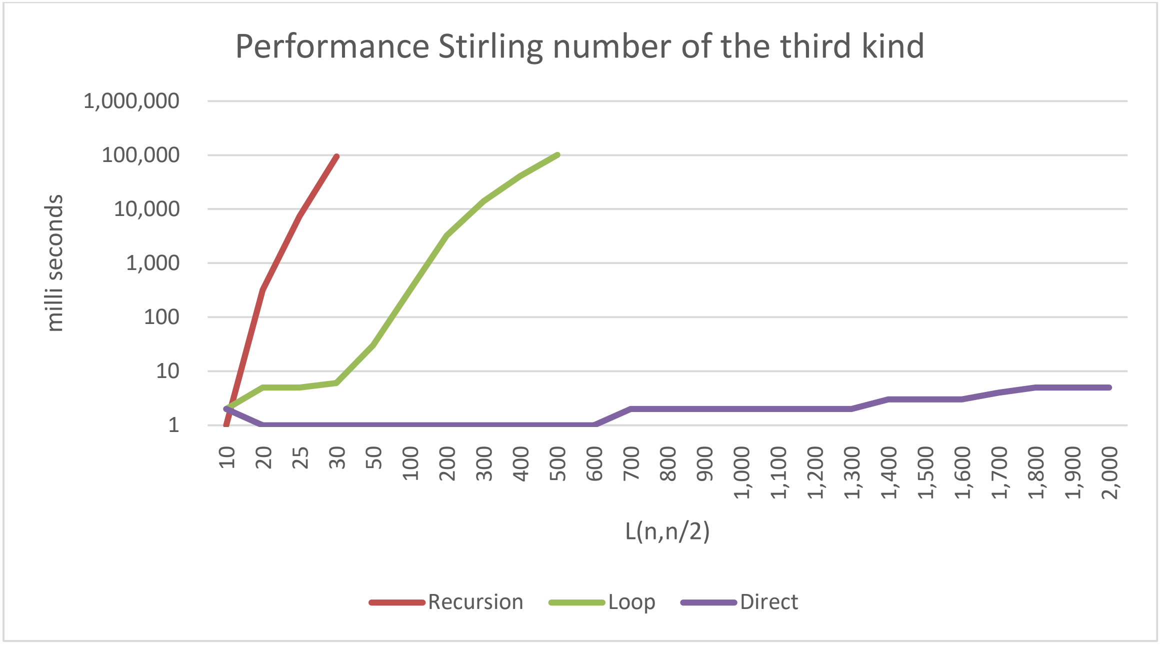

Figure 3. Performance of the Stirling number of the third kind.

From the chart above it is very clear that the recursive method is performing poorly compared to the two others. The Loop method is the summation of both the Stirling number of the first and second kind and it also has a terrible performance characteristic. Surprisingly the direct formula is superior to the two others method and scales well with the higher number of n.

Recommendation for the Stirling number of the third kind

I recommend the following:

To use the direct formula (12) and (13) for the fastest performance.

Overall recommendation for all three kinds.

Based on the performance the recursive solution to find the Stirling number of the first, second, or third kind is very popular but doesn’t perform and scales well with the higher number of n and therefore should be avoided in use with arbitrary precisions. Instead, use the recommended method for the three different kinds of Stirling numbers.

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages
2) Wikipedia. Stirling number. Stirling number - Wikipedia
3) Wikipedia. Stirling number of the first kind. Stirling numbers of the first kind - Wikipedia
4) Wikipedia. Stirling number of the second kind. Stirling numbers of the second kind - Wikipedia
5) Wikipedia. Lah number. Lah number - Wikipedia
6) J. Arndt. Matters Computational, Ideas, Algorithms, Source code, www.jjj.de/fxt/fxtpage.html#fxtbook
7) ChatGPT (www.openai.com) on March 5, 2023