

Fast Factorials & Binomial Computation for Arbitrary Precision

By Henrik Vestermark (hve@hvks.com)

Abstract:

This paper continues a series addressing practical challenges in constructing an arbitrary-precision arithmetic toolkit. It focuses on the computation of factorials, falling and rising factorials, and binomial coefficients under arbitrary-precision constraints. Although these functions are often regarded as elementary and are commonly presented via simple recursive definitions, such approaches scale poorly in high-precision settings. As operand sizes grow, algorithmic structure, multiplication balance, and control overhead become decisive factors for performance.

The paper examines and compares several factorial computation strategies, including optimized loop-based methods, odd-only decomposition, prime swing, and binary splitting, with particular emphasis on their behavior in arbitrary-precision arithmetic. Empirical performance measurements are used to identify practical crossover points and to motivate hybrid implementations. Based on these results, the paper provides concrete guidance on algorithm selection and implementation techniques for efficient and scalable arbitrary-precision computation of factorial-related functions.

Introduction:

We start with simple functions such as factorials, falling and rising factorials, and binomial coefficients, discuss the performance of the various methods, and recommend which method to use for arbitrary-precision computations.

All of these simple functions can be implemented using arbitrary-precision integer arithmetic. Furthermore, the paper presents a systematic performance comparison of factorial algorithms in an arbitrary-precision setting, practical hybrid implementations with threading, and empirically derived crossover points to guide algorithm selection. As customary, the actual C++ source code for the calculations will be provided, using the author's arbitrary-precision math library [1].

This paper is part of a series of documents on arbitrary precision, 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 and 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](#)

Fast Factorial & Binomial Computation 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 an 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](#)
12. Fast Computation of Stirling's numbers in arbitrary precision. [HVE Fast Computation of Stirling numbers in arbitrary precision](#)
13. Fast Prime computation in arbitrary precision. [HVE Fast Prime Computation in arbitrary precision](#)
14. Fast PRNG in arbitrary precision. [HVE Fast PRNG in arbitrary precision](#)
15. Fast Fibonacci sequence in arbitrary precision. [HVE Fast Fibonacci in arbitrary precision](#)
16. Fast arbitrary precision Multiplication. [HVE Fast arbitrary precision integer multiplication](#)
17. Fast Factorial and Binomial computation with arbitrary precision. [HVE Fast factorial and binomial for arbitrary precision.docx](#)

Fast Factorial & Binomial Computation for Arbitrary Precision

Contents

Abstract:	1
Introduction:.....	1
The Arbitrary precision library	4
Int_precision class.....	4
Internal format for int_precision variables	4
The History of Factorials	5
Applications for factorials.....	5
Factorials.....	6
Factorial using recursion.....	7
Source factorial via recursion	7
Factorial using the loop-based algorithm.....	7
Source factorial looping.....	7
Switching to arbitrary precision for large factorials	8
Source factorial looping using arbitrary precision.....	8
Source factorial loop balance.....	8
Source factorial loop balance max.....	8
Odd-Only Product with Power-of-Two Reattachment	9
Source code factorial odd product with shift.....	10
Factorial Prime Swing.....	11
The method can be mathematically described (simplified).....	11
Source code for factorial prime swing.....	12
Factorial using binary splitting	14
Source factorial binary splitting.....	15
Performance of Factorials	16
Performance of factorial algorithms	17
Recommendation for factorial computation	17
Factorial using a hybrid approach.....	18
Source Hybrid factorial with threading.....	18
Falling Factorials	19
Source falling factorial with arbitrary precision.....	20
Falling Factorial using binary splitting.....	21
Source falling factorial hybrid implementation.....	21
Performance of falling factorial.....	23
Rising Factorial.....	23
Binomial coefficients.....	23
Source for binomial	24
Reference	24

Fast Factorial & Binomial Computation for Arbitrary Precision

The Arbitrary precision library

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

Int_precision class

To understand the C++ code and text, we highlight a few features of the arbitrary-precision library, specifically the *int_precision* class. Instead of declaring a variable with any of the built-in integer types `char`, `short`, `int`, `long`, `long long`, `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`, you 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 integer in C++. Furthermore, there are a few methods you will need to know.

One of them is `.iszero()`, which returns true or false depending on whether the *int_precision* variable is zero. Another is `.even()` and `.odd()`, which return the Boolean values for even and odd status. There are other methods, but I 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 integers 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. `mNumber[size()-1]` contains the most significant 64-bit component. The sign is stored separately in the class field variable `mSign`, so `mNumber` holds the unsigned binary vector data.

For more details, see [1].

The History of Factorials

The factorial, denoted by an exclamation point, $!$, is a fundamental concept in mathematics, especially in combinatorics, algebra, and mathematical analysis. The history of factorials is closely linked to the study of permutations and combinations. Here's a brief history:

1. *Ancient Times*: The idea of factorial can be traced back to ancient civilizations. The Indians and Greeks studied permutations and combinations, which inherently involve factorial calculations, even if they didn't use modern notation or names.
2. *Middle Ages*: In the 1200s, scholars gave descriptions related to the concept of factorial when discussing the number of possible permutations of the Hebrew Bible's verses.
3. *Early Modern Period*: Swiss mathematician Leonhard Euler is often credited with introducing and popularizing the modern notation " $n!$ " for factorial in the 1700s. By this time, the product definition of factorials was well-understood, i.e., $n! = n(n-1) \dots (2)(1)$.
4. *18th and 19th Centuries*: Factorial became increasingly significant in the areas of analysis, combinatorics, and probability theory. Notably, factorials play a crucial role in the development of the binomial theorem and the definition of the exponential function.
5. *20th Century to Present*: Factorials continued to be of importance, especially in the study of series and sequences, generating functions, and the analysis of algorithms in computer science. Moreover, the factorial concept was extended to non-integer values using the gamma function.

Factorials are a fundamental concept taught in schools and are widely used across mathematics, physics, and engineering.

Applications for factorials

Factorials are deeply embedded in many areas of mathematics and their applications. Here's a more detailed look at some of their primary uses:

1. *Combinatorics and Permutations*: Permutations are the number of ways to arrange n distinct items in a specific order is $n!$. Combinations are when choosing r items out of n without replacement, and when the order doesn't matter, the formula is $\frac{n!}{r!(n-r)!}$.
2. *Probability*: Factorials play a pivotal role in determining the number of possible outcomes in various probabilistic scenarios. For example, the probability of a specific sequence happening in a set number of events can often be calculated using factorials.
3. *Binomial Theorem*: When expanding $(x + y)^n$. The coefficients of the terms are combinations (which involve factorials). $\binom{n}{m} = \frac{n!}{m!(n-m)!}$. This relates to Pascal's Triangle.
4. *Calculus*: Taylor and Maclaurin Series. Factorials are used in the expansion of functions as an infinite series and differential Equations. Some solutions, especially for certain recursive relationships, involve factorials.
5. *Quantum Mechanics*: In quantum statistics, factorials can appear in the calculations of the number of ways particles can be arranged.
6. *Computer Science*: Particular algorithm Analysis where factorials can be used to describe the worst-case scenarios of specific algorithms, especially when looking at all

Fast Factorial & Binomial Computation for Arbitrary Precision

possible configurations or orderings, or in data Structures, where specific tree structures or recursive data structures might have relationships best described using factorials.

7. *Biology*: Population models and genetics sometimes use factorials when determining possible gene combinations or looking at potential evolutionary pathways.

8. *Number Theory*: Factorials are involved in various number-theoretical problems, including properties of prime numbers and combinatorial number theory.

9. *Gamma Function*: This is a generalization of the factorial, allowing us to compute "factorials" of non-integer values (excluding negative integers).

These are just a few of the areas where factorials arise. They are everywhere in mathematical applications, underscoring their fundamental and versatile nature.

Factorials

Factorials are fundamental mathematical operations frequently employed in various applications. They are commonly expressed as either an iterative or a recursive formula.

The iterative formula is as follows:

$$n! = n \cdot (n - 1) \cdot (n - 2) \dots 2 \cdot 1 = \prod_{k=1}^n k \quad (1)$$

Or the recursive formula:

$$n! = n \cdot (n - 1)! \quad (2)$$

Where $0!=1$ and $1!=1$.

However, when implemented using basic data types such as integers, factorials can overflow quickly.

Even when running the code above on a 64-bit system, you get an overflow for factorials greater than $20!$ This limitation can be addressed by using arbitrary-precision arithmetic libraries to handle larger values.

This is one of the many situations where an arbitrary-precision integer math library comes in handy to save the day. [1]

You can usually compute factorials using one of the basic methods below, with or without variations and optimization techniques.

- 1) *Factorial using recursion*. A straightforward recursive approach is to compute factorials as shown in the code snippet. This method, while simple, may be inefficient due to repeated calculations.
- 2) *Factorial using looping*. Loop-based implementations are often preferred over recursion due to better efficiency. The given loop-based code computes factorials iteratively, reducing the risk of stack overflow and improving performance.
- 3) *Factorial using Odd-only product plus shifting*.
- 4) *Factorial using the Prime Swing method*.
- 5) *Factorial using binary splitting*. The binary splitting method offers an efficient alternative for calculating large factorials. It recursively splits the computation into two halves and then multiplies the results. This approach is particularly advantageous when using arbitrary precision arithmetic. The provided code outlines the binary-splitting algorithm and its performance benefits.

Fast Factorial & Binomial Computation for Arbitrary Precision

There are others, but usually not entirely relevant for an exact computation of factorial. Here, we can mention Stirling's approximation, which, for large n , gives an estimate of $n!$ that becomes more accurate as n grows. The formula is:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \quad (3)$$

While this doesn't yield the exact value, it provides a close approximation for large n , which is sufficient in many applications.

For non-integer values, the factorial can be generalized using the gamma function, denoted $\Gamma(n)$ Where:

$$\Gamma(n) = (n - 1)! \quad (4)$$

There are various methods for computing values of the gamma function, such as continued fractions or series expansions. But these are more complex than the simple factorial definition.

Depending on the scenario (e.g., n , required precision, computational resources), one may choose the most appropriate method for computing the factorial.

Factorial using recursion

Most often, you see it implemented as straightforward recursive code.

Source factorial via recursion

```
uintmax_t factorial(uintmax_t k)
{
    if (k <= 1)
        return 1;
    return k * factorial(k - 1);
}
```

Factorial using the loop-based algorithm

As we were told in computer science class, recursion is good, but unrolling it into a loop-based algorithm is better. The above code, rewritten into a more efficient loop-based implementation, is:

Source factorial looping

```
uintmax_t factorialloop(uintmax_t k)
{
    uintmax_t res = 1;
    for (; k > 1; --k)
        res *= k;
    return res;
}
```

Fast Factorial & Binomial Computation for Arbitrary Precision

Switching to arbitrary precision for large factorials

To overcome the limitation of a maximum of 20! (unsigned data type in a 64-bit system), You need to switch to arbitrary-precision integer arithmetic.

The loop-based source above looks like this in the author's arbitrary-precision math package, where the type for an integer is called *int_precision*. [1]

Source factorial looping using arbitrary precision

```
int_precision factorialloop(const int_precision& kip)
{
    uintmax_t k = (uintmax_t)kip;
    int_precision res = int_precision(1);
    for (; k > 1; --k)
        res *= int_precision(k);
    return res;
}
```

Switching to arbitrary precision allows you to go to "unlimited" factorials; however, the above algorithm is not particularly fast since arbitrary precision slows down the computation. One improvement is to perform some intermediate computations using 64-bit arithmetic. E.g., we could group two consecutive numbers and then add this number to an arbitrary-precision result; to limit overflow, we could rearrange the factorial and start by multiplying the top *n* by 1, then (*n*-1) by 2, etc.

$$n! = n \cdot 1 \cdot (n - 1) \cdot 2 \dots \left(\frac{n}{2}\right) \left(\frac{n}{2} - 1\right) \quad (5)$$

This has the benefit that the biggest number will be less than $\left(\frac{n}{2}\right)^2$ Limiting the possibility of overflow. The arbitrary precision implementation will be as follows:

Source factorial loop balance

```
int_precision factorialloopbalance(const int_precision& kip)
{
    uintmax_t l = 1, k=(uintmax_t)kip;
    int_precision res = int_precision(1);
    for (; k > l; --k, ++l)
        res *= int_precision(k*l);
    if(k==l)
        res *= int_precision(k);
    return res;
}
```

This provides a significant speedup because the pairwise multiplication of *k* and *l* is performed using 64-bit arithmetic. Continue along that idea; you can generalize it by performing pairwise multiplication multiple times, depending on the range of numbers you are multiplying.

Source factorial loop balance max

```
int_precision factorial(const int_precision& kip)
{
    const uintmax_t UINTMAX_T_MAX= ~((uintmax_t)0);
    uintmax_t m=1, kk, prod, k=(uintmax_t)kip;
    int_precision res = int_precision(1);

    if (k <= 1)
        return res;
}
```

Fast Factorial & Binomial Computation for Arbitrary Precision

```
for (kk = 1; k>m; --k, ++m)
{
    prod = k * m;          // Never overflow
    if (kk < UINTMAX_T_MAX / prod)
        kk *= prod;
    else
    {
        res *= int_precision(kk);
        kk = prod;
    }
}
res *= int_precision(kk);
if (k == m)
    res *= int_precision(k);
return res;
}
```

The performance improvement from the simple loop-based version to the more advanced version above is a factor of 6-7. With arbitrary-precision arithmetic, you always want to prioritize speed over simplicity.

Odd-Only Product with Power-of-Two Reattachment

The Origin and historical context of the odd-only product with power-of-two reattachment is not a single named algorithm. Still, a classical decomposition of the factorial based on separating powers of two. Its mathematical foundation relies on the formula for the exponent of a prime in a factorial, commonly attributed to Legendre in the early 19th century:

$$v_2(n!) = \sum_{k=1}^{\infty} \left\lfloor \frac{n}{2^k} \right\rfloor \quad (6)$$

This identity allows the factorial to be expressed as:

$$n! = 2^{v_2(n!)} \cdot \text{odd}(n!) \quad (7)$$

Where $\text{odd}(n!)$ denotes the product of all factors of $n!$ with their powers of two removed. The odd product computation still has $O(n)$ multiplicative complexity, just with fewer large multiplications. The algorithmic use of this decomposition emerged in the late 20th century with the development of fast arbitrary-precision arithmetic.

A simplified Mathematical description is shown below.

Every integer $k \geq 1$ can be written uniquely as:

$$k = 2^{v_2(k)} \cdot k_{\text{odd}} \quad (8)$$

Applying this factorization to all terms in the factorial gives:

$$n! = \prod_{k=1}^n 2^{v_2(k)} \cdot \prod_{k=1}^n k_{\text{odd}} \quad (9)$$

Rearranging terms yields:

Fast Factorial & Binomial Computation for Arbitrary Precision

$$n! = 2^{v_2(n!)} \cdot \prod_{k=1}^n k_{\text{odd}} \quad (10)$$

The exponent $v_2(n!)$ is computed using Legendre's formula, while the odd part is constructed by collecting all odd contributions. While the number of multiplications is reduced, the odd-only method does not asymptotically improve over balanced product trees.

An example will clarify the formula. Let's compute 10!

Step 1: Determine the power of two

$$v_2(10!) = \lfloor 10/2 \rfloor + \lfloor 10/4 \rfloor + \lfloor 10/8 \rfloor = 5 + 2 + 1 = 8$$

Thus: $10! = 2^8 \cdot \text{odd}(10!)$

Step 2: Determine the odd part

For successive powers of two:

$$k = 1: \lfloor 10/1 \rfloor = 10 \rightarrow \{1, 3, 5, 7, 9\}, \text{ product} = 945$$

$$k = 2: \lfloor 10/2 \rfloor = 5 \rightarrow \{1, 3, 5\}, \text{ product} = 15$$

$$k = 4: \lfloor 10/4 \rfloor = 2 \rightarrow \{1\}, \text{ product} = 1$$

$$k = 8: \lfloor 10/8 \rfloor = 1 \rightarrow \{1\}, \text{ product} = 1$$

Multiplying these contributions gives:

$$\text{odd}(10!) = 945 \cdot 15 = 14175$$

Step 3: reattach powers of two

$$10! = 14175 \cdot 2^8 = 14175 \cdot 256 = 3628800$$

Source code factorial odd product with shift

```
int_precision factorialodd_shift(const int_precision& ipn)
{
    uint64_t n = static_cast<uint64_t>(ipn);
    if (n < 2) return int_precision(1);

    // portable popcount
    auto popcount_u64 = [](uint64_t x) -> uint64_t {
        uint64_t c = 0;
        while (x) { x &= x - 1; ++c; }
        return c;
    };

    auto power_of_two_factor = [](uint64_t x) -> uint64_t {
        return x - popcount_u64(x); // e2(x!) = x - popcount(x)
    };

    // product of odd numbers in [b, a], inclusive
    auto odd_product = [&](auto&& self, uint64_t a, uint64_t b) -> int_precision {
        if (a < b) return int_precision(1);

        if (a == b) {
            return (a & 1) ? int_precision(a) : int_precision(1);
        }

        uint64_t diff = a - b;
        if (diff == 1) {
            int_precision r(1);
            if (a & 1) r *= int_precision(a);
            if (b & 1) r *= int_precision(b);
            return r;
        }

        uint64_t m = (a + b) >> 1;
        return self(self, a, m) * self(self, m - 1, b);
    };
}
```

Fast Factorial & Binomial Computation for Arbitrary Precision

```
auto odd_productalternative = [&](auto&& self, uint64_t a, uint64_t b) ->
int_precision {
    if (a < b) return int_precision(1);
    if (a == b) return (a & 1) ? int_precision(a) : int_precision(1);

    uint64_t diff = a - b;

    // Direct multiplication for small ranges
    if (diff <= 64) { // Tune this threshold
        int_precision r(1);
        if (!(b & 1)) ++b; // Start at odd
        for (uint64_t i = b; i <= a; i += 2) {
            r *= int_precision(i);
        }
        return r;
    }

    uint64_t m = (a + b) >> 1;
    return self(self, a, m) * self(self, m - 1, b);
};

auto odd_factorial_part = [&](uint64_t x) -> int_precision {
    int_precision result(1);
    for (uint64_t k = 1; k <= x; k <<= 1) {
        uint64_t high = x / k;
        if (high < 1) break;
        if ((high & 1) == 0) --high; // highest odd <= high
        if (high >= 1) result *= odd_product(odd_product, high, 1);
    }
    return result;
};

int_precision odd = odd_factorial_part(n);
odd <<= power_of_two_factor(n);
return odd;
}
```

Factorial Prime Swing

Peter Luschny introduced the prime swing factorial algorithm in the early 2000s as part of his work on fast exact computation of factorials for large integers. The method builds on classical number theoretic results, particularly Legendre's formula for the exponent of a prime in a factorial, but reorganizes the computation to minimize the number of large multiplications. Today, prime swing is widely regarded as one of the fastest exact factorial algorithms and is often used as a benchmark for high-performance arbitrary-precision implementations.

The method can be mathematically described (simplified)

As with the odd-only product method, the factorial is decomposed into an odd part and a power of two:

$$n! = 2^{v_2(n!)} \cdot \text{oddFact}(n) \quad (11)$$

The odd part is computed recursively using the identity:

$$\text{oddFact}(n) = \text{oddFact}(\lfloor \frac{n}{2} \rfloor)^2 \cdot \text{swing}(n) \quad (12)$$

Fast Factorial & Binomial Computation for Arbitrary Precision

The swing function collects the remaining odd prime factors appearing in the quotient:

$$\frac{n!}{\left(\lfloor \frac{n}{2} \rfloor!\right)^2} \quad (13)$$

Excluding the prime 2. For each odd prime $p \leq n$, the exponent is given by:

$$e_p = v_p(n!) - 2 \cdot v_p(\lfloor \frac{n}{2} \rfloor!) \quad (14)$$

The swing term is then defined as:

$$\text{swing}(n) = \prod_{\substack{p \leq n, \\ p \text{ odd prime}}} p^{e_p} \quad (15)$$

The prime swing reduces the problem to roughly $O(n \log n)$ bit complexity, assuming a fast multiplication method (FFT or NTT).

An example will clarify the formula. Let's compute $10!$ again

Step 1: Determine the power of two using Legendre's formula.

$$v_2(10!) = \lfloor 10/2 \rfloor + \lfloor 10/4 \rfloor + \lfloor 10/8 \rfloor = 8$$

Thus: $10! = 2^8 \cdot \text{oddFact}(10)$

Step 2: Compute the odd factorial recursively.

$$\text{oddFact}(10) = \text{oddFact}(5)^2 \cdot \text{swing}(10)$$

$\text{oddFact}(5) = 15$ (see the interim computation in the previous example).

The relevant odd primes are 3, 5, and 7. Their exponents are:

$$e_3 = 4 - 2 \cdot 1 = 2$$

$$e_5 = 2 - 2 \cdot 1 = 0$$

$$e_7 = 1 - 0 = 1$$

Hence: $\text{swing}(10) = 3^2 \cdot 7 = 63$

Now compute the odd part: $\text{oddFact}(10) = 15^2 \cdot 63 = 14175$

Step 3: reattach the power of two.

$$10! = 14175 \cdot 2^8 = 3628800$$

Source code for factorial prime swing

```
int_precision factorial_prime_swing(const int_precision& ipn)
{
    const uint64_t n = static_cast<uint64_t>(ipn);
    if (n < 2) return int_precision(1);

    // popcount (portable, C++17 friendly)
    auto popcount_u64 = [](uint64_t x) -> uint64_t {
        uint64_t c = 0;
        while (x) { x &= x - 1; ++c; }
        return c;
    };

    // exponent of 2 in n! : e2(n!) = n - popcount(n)
    auto e2_factorial = [](uint64_t x) -> uint64_t {
        return x - popcount_u64(x);
    };
}
```

Fast Factorial & Binomial Computation for Arbitrary Precision

```
// sieve primes up to n (simple sieve, replace with segmented sieve if n is large)
auto sieve_primes = [&](uint64_t limit) -> std::vector<uint32_t> {
    std::vector<bool> is_prime(limit + 1, true);
    if (limit >= 0) is_prime[0] = false;
    if (limit >= 1) is_prime[1] = false;

    for (uint64_t p = 2; p * p <= limit; ++p) {
        if (!is_prime[p]) continue;
        for (uint64_t k = p * p; k <= limit; k += p) is_prime[k] = false;
    }

    std::vector<uint32_t> primes;
    primes.reserve(static_cast<size_t>(limit / 10 + 10));
    for (uint64_t i = 2; i <= limit; ++i)
        if (is_prime[i]) primes.push_back(static_cast<uint32_t>(i));
    return primes;
};

// v_p(n!) (Legendre)
auto v_p_factorial = [&](uint64_t x, uint32_t p) -> uint64_t {
    uint64_t e = 0;
    while (x) { x /= p; e += x; }
    return e;
};

// small base exponentiation into int_precision
auto pow_small = [&](uint32_t base, uint64_t exp) -> int_precision {
    int_precision result(1);
    int_precision b(base);
    while (exp) {
        if (exp & 1) result *= b;
        exp >>= 1;
        if (exp) b *= b;
    }
    return result;
};

const std::vector<uint32_t> primes = sieve_primes(n);

// swing(n) =  $\prod_{\text{odd primes } p \leq n} p^{\{v_p(n!) - 2 * v_p((n/2)!)\}}$ 
auto swing = [&](uint64_t x) -> int_precision {
    const uint64_t m = x >> 1;
    int_precision s(1);

    for (uint32_t p : primes) {
        if (p > x) break;
        if (p == 2) continue;

        const uint64_t e = v_p_factorial(x, p) - 2ull * v_p_factorial(m, p);
        if (e) s *= pow_small(p, e);
    }
    return s;
};

// oddFact recursion: oddFact(n) = oddFact(n/2)^2 * swing(n)
auto odd_fact = [&](auto&& self, uint64_t x) -> int_precision {
    if (x < 2) return int_precision(1);
    int_precision r = self(self, x >> 1);
    r *= r;
    r *= swing(x);
    return r;
};

int_precision odd = odd_fact(odd_fact, n);
odd <<= e2_factorial(n);
return odd;
}
```

Fast Factorial & Binomial Computation for Arbitrary Precision

Factorial using binary splitting

One method needs to be explained: binary splitting. Generally, with the balance loop-based approach, you are fine. Still, if you are dealing with a very large factorial, the binary splitting method will perform better than any previous method described, except for the prime swing, which has a similar performance characteristic to the other methods. The binary splitting method computes a product over the interval $[m+1, n]$, which reduces to $n!$ when $m=0$. The binary splitting method recursively splits the computation into two halves, calls itself on each half, and then multiplies the two halves. In essence, it performs the same number of multiplications as the other methods but is more efficient, particularly when using arbitrary-precision integer arithmetic. The binary splitting method divides two factorials. $\frac{n!}{m!}$. To find $n!$ You set $m=0$. The algorithm is as follows and is called with two integer parameters n and m .

```
Factorial_binary_splitting(n,m)
  If(n-1=m) return n;
  mid=(n+m)/2
  high = Factorial_binary_splitting(n,mid)
  low = Factorial_binary_splitting(mid,m)
  Return high * low
```

$n!$ is computed with the call `FactorialBinarySplitting(n,0);`

Example for $10!$.

```
Bs(10,0)
  Bs(10,5)
    Bs(10,7)
      Bs(10,8)
        Bs(10,9) return 10
        Bs(9,8) return 9
      Return 90
    Bs(8,7) return 8
  Return 720
Bs(7,5)
  Bs(7,6) return 7
  Bs(6,5) return 6
Return 42
Return 30240
Bs(5,0)
  Bs(5,2)
    Bs(5,3)
      Bs(5,4) return 5
      Bs(4,3) return 4
    Return 20
  Bs(3,2) return 3
Return 60
Bs(2,0)
```

Fast Factorial & Binomial Computation for Arbitrary Precision

```
Bs(2,1)  return 2
Bs(1,0)  return 1
Return 2
Return 120
Return 3628800
```

The recursion ensures that all large multiplications are between numbers of similar size, minimizing multiplication cost.

To reduce the number of recursions, you return as soon as there are only 2-3 elements different to avoid costly recursions. In the source code below, it returns when the difference is three or lower.

Source factorial binary splitting

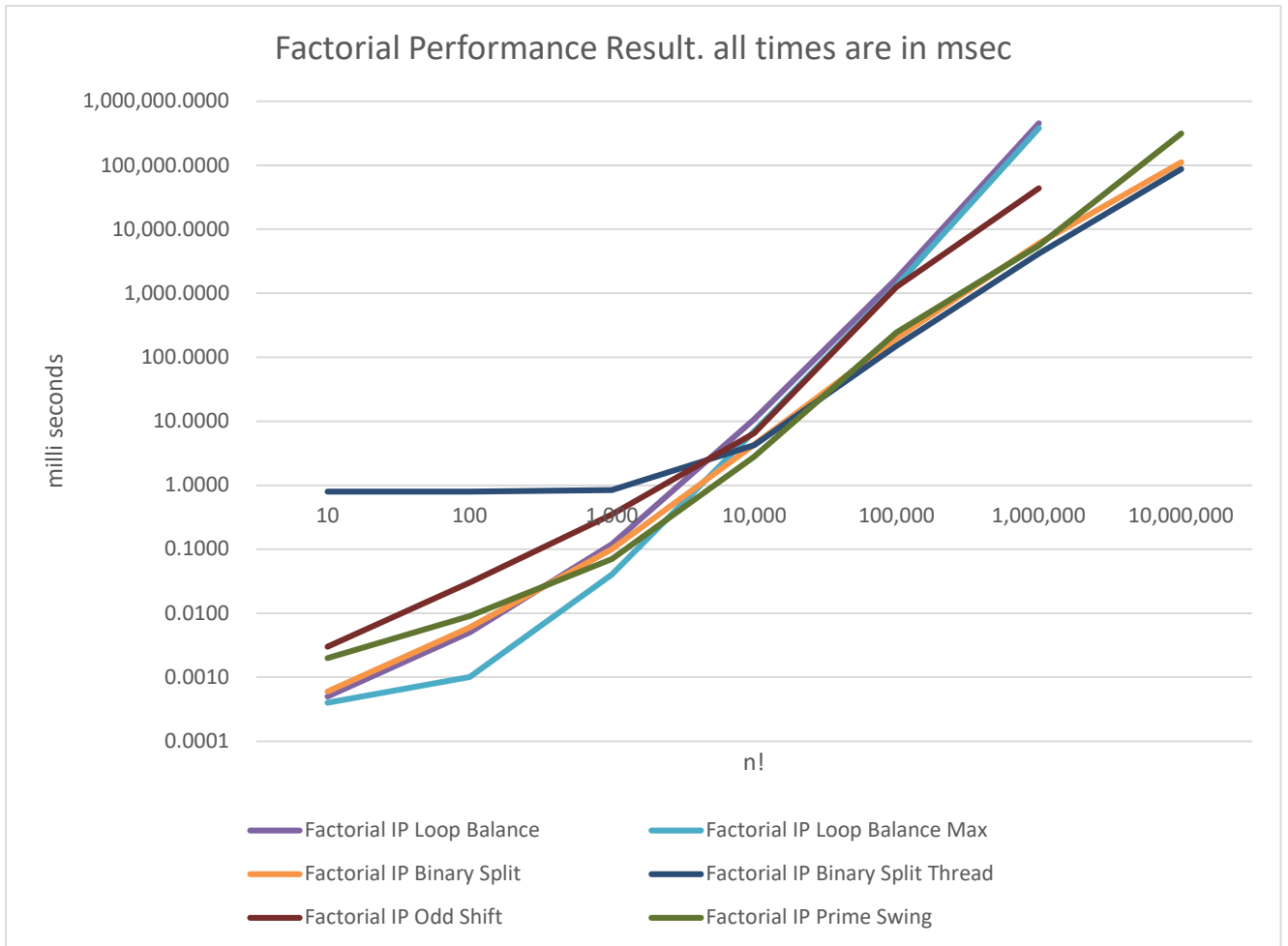
```
int_precision factorial_binary_splitting(const int_precision& a, const
int_precision& b)
{
    const int_precision c1(1), c2(2);
    const int_precision diff = a - b;

    switch (static_cast<uintmax_t>(diff))
    {
        // Base cases
        case 0: return c1; break;
        case 1: return a; break;
        case 2: return a * (a - c1); break;
        case 3: return a * (a - c1) * (a - c2); break;
        default:// Fall through
            break;
    }
    int_precision m = (a + b) / c2;
    return factorial_binary_splitting(a, m) * factorial_binary_splitting(m, b);
}
```

To further improve performance, you can add threading to the binary-splitting method, which typically increases performance by 30-50%. Because the binary splitting method is recursive, it is straightforward to multithread.

Fast Factorial & Binomial Computation for Arbitrary Precision

Performance of Factorials



All benchmarks were performed on a single workstation using GCC/MSVC with optimization enabled.

Below is the actual timing in milliseconds.

Factorial Performance Result. All times are in msec.							
Digits	10	100	1,000	10,000	100,000	1,000,000	10,000,000
Factorial IP	0.0040	0.040	0.61	27.5			
Factorial IP Loop Balance	0.0005	0.005	0.12	10.7	1,693	453,289	
Factorial IP Loop Balance Max	0.0004	0.001	0.04	7	1,303	382,861	
Factorial IP Binary Split	0.0006	0.006	0.10	4.3	197	6,010	111,739
Factorial IP Binary Split Thread	0.8000	0.800	0.84	4.2	150	4,171	87,110
Factorial IP Odd Shift	0.0030	0.030	0.35	6.5	1,251	44,008	
Factorial IP Prime Swing	0.0020	0.009	0.07	2.8	242	5,586	315,680

Fast Factorial & Binomial Computation for Arbitrary Precision

Performance of factorial algorithms

The performance measurements in the table above compare several factorial algorithms across a wide range, from $10!$ up to $10,000,000!$, using arbitrary-precision integer arithmetic. The results demonstrate that control overhead dominates for very small inputs, while multiplication balance and algorithmic structure become decisive as the factorial size grows.

The naive loop-based approach, even when enhanced by loop balancing, performs adequately only for small factorials and scales poorly as the number of digits increases. The optimized loop-balance-max variant significantly improves performance by maximizing 64-bit arithmetic and is the fastest method for very small factorials, up to approximately $1,000!$. Beyond this range, its performance degrades rapidly relative to more structured algorithms.

The binary splitting method consistently outperforms loop-based approaches once factorial sizes exceed a few thousand. By recursively splitting the product into balanced subranges, binary splitting minimizes the cost of large, unbalanced multiplications and makes efficient use of fast arbitrary-precision multiplication algorithms. The advantage of this structure becomes increasingly pronounced as the factorial size grows.

When combined with multithreading, binary splitting achieves the best overall performance for large factorials. Although threading introduces noticeable overhead for very small inputs, it becomes beneficial around $10,000!$, and from approximately $100,000!$ moving forward, the threaded binary-splitting implementation clearly dominates. This dominance persists through $1,000,000!$ and up to $10,000,000!$, where no other method in the current implementation approaches its performance.

The odd-only product with power-of-two reattachment reduces the number of multiplications by eliminating even factors and handling powers of two via a single shift. While mathematically elegant and conceptually straightforward, the measurements show that this method does not compete with binary splitting or prime swing for large factorials in the present implementation. Its primary value lies in its pedagogical clarity and its role as an intermediate optimization technique.

The prime swing method demonstrates strong performance across a broad mid-range of inputs. In particular, it is the fastest method around $10,000!$, outperforming both loop-based approaches and single-threaded binary splitting. However, as factorial sizes approach and exceed $100,000!$, the overhead associated with prime generation, exponent bookkeeping, and power construction becomes increasingly visible. In this range and beyond, the simpler control flow of binary splitting, especially in its multithreaded form, exhibits a clear and growing performance advantage. At $1,000,000!$ and $10,000,000!$, threaded binary splitting substantially outperforms prime swing in the current implementation.

Recommendation for factorial computation

Based on the corrected and extended measurements, the following strategy is recommended for arbitrary-precision factorial computation:

- For very small factorials (up to approximately $1,000!$), the optimized loop-balance-max method provides the best performance with minimal complexity.

Fast Factorial & Binomial Computation for Arbitrary Precision

- For moderate factorial sizes (roughly 1,000! to 20,000!), structured methods dominate; prime swing is highly competitive and can be the fastest method, particularly around 10,000!.
- For large factorials (from approximately 100,000! and upward), binary splitting with multithreading offers the best and most consistent performance and is the recommended default method in the current library.
- The odd-only product with power-of-two reattachment remains valuable from a theoretical and instructional perspective. Still, it is not recommended as a primary method for large factorials in this implementation.

While prime swing remains one of the most efficient known factorial algorithms from an asymptotic standpoint, practical performance is strongly influenced by implementation overhead and control complexity. For the present arbitrary-precision library and the measured range up to 10,000,000!, a hybrid approach that ultimately favors multithreaded binary splitting provides the most reliable and efficient solution.

Factorial using a hybrid approach

The hybrid approach combines the benefits of loop-based and binary splitting methods. It utilizes looping for smaller values of n and switches to binary splitting for larger values, offering an optimal balance of performance and accuracy. For optimal results, Multithreading can be employed to enhance the performance of the binary further splitting method.

We can now present the hybrid factorial solutions that take advantage of both the loop-based approach and the binary splitting method, and include threading for optimal performance.

The threshold values were determined empirically and may vary with hardware and compiler environment.

Source Hybrid factorial with threading.

```
int_precision factorial_hybrid_thread(const int_precision& n )
{
    const int_precision c0(0), c1(1), c2(2);

    auto factorial_loop = [&](const int_precision& n)
    {
        const uintmax_t UINTEMAX_T_MAX = ~(0ull);
        uintmax_t m = 1, kk, prod, k = static_cast<uintmax_t>(n);
        int_precision res = c1;

        if (k <= 1)
            return res;
        for (kk = 1; k > m; --k, ++m)
        {
            prod = k * m;          // Never overflow
            if (kk < UINTEMAX_T_MAX / prod)
                kk *= prod;
            else
            {
                res *= int_precision(kk);
                kk = prod;
            }
        }
        res *= int_precision(kk);
        if (k == m)
```

Fast Factorial & Binomial Computation for Arbitrary Precision

```
        res *= int_precision(k);
        return res;
    };

    // We need to use the std::function to be able to call the lambda
    // function recursively.
    std::function<int_precision(const int_precision&, const int_precision&>
factorial_binary_splitting = [&](const int_precision& a, const int_precision& b)
    {
        const int_precision diff = a - b;

        switch (static_cast<uintmax_t>(diff))
        { // Base cases
        case 0: return c1; break;
        case 1: return a; break;
        case 2: return a * (a - c1); break;
        case 3: return a * (a - c1) * (a - c2); break;
        default:// Fall through
            break;
        }
        int_precision m = (a + b) / c2;
        return factorial_binary_splitting(a, m) * factorial_binary_splitting(m, b);
    };

    // The balance loop base is fastest below 5000!
    if (n <= int_precision(5000))
        return factorial_loop(n);

    // Do the factorial binary splitting method above 5000!
    // with or without threading. 2 ways Threading improves the speed by 30-60% over
    // the non-threaded version
    // This, if needed, can easily be expanded to 3-4 or n-ways threading to
    // improve performance
    int_precision m = (n) / c2, high, low;
    std::thread first([&]()
    {
        high = factorial_binary_splitting(n, m);
    }); // interval [n...m]
    std::thread second([&]()
    {
        low = factorial_binary_splitting(m, c0);
    }); // interval [m...0]
    first.join();
    second.join();

    return high * low;
}
```

Falling Factorials

The falling factorial describes the factorial between two numbers m and n in falling (or descending notation):

$$n^{\underline{m}} = n \cdot (n - 1) \cdot (n - 2) \dots (n - m + 1) = \prod_{k=n-m+1}^n k \quad (16)$$

Falling factorials are useful when calculating the binomial coefficient:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \quad (17)$$

Since the falling factorial is:

Fast Factorial & Binomial Computation for Arbitrary Precision

$$\frac{n!}{(n-m)!} = n^m \quad (18)$$

It can be handled with a single call to the falling factorial function rather than two calls to the factorial function.

See the next chapter for the computation of the binomial coefficient.

The falling factorial has two parameters, n and m , whereas the regular factorial has only one. We again optimized computation by performing intermediate steps in 64-bit arithmetic. In the example below, we specify the parameters n and m in the definition and use the shortcut that n and m are limited to $2^{64}-1$ or fit within an unsigned 64-bit integer. This limitation is a reasonable choice in nearly all cases. Since we try to perform as many calculations as possible using 64-bit integers, we must build safeguards against unintended overflow. We know from the factorial chapter that recursive function calls are not as fast as loop-based computation, so we go straight to the loop-based solution with arbitrary precision.

Source falling factorial with arbitrary precision

```
// Handles falling factorial
// Same as n!/(n-m)! == (n-m+1)*(n-m+2)*...*(n-1)*n
// limitation: n must fit in uint64_t, not entirely arbitrary precision
// but a reasonable max limit
// m is always less than n but no domain error is thrown if it is not
int_precision fallingfactorial(const int_precision& n, const int_precision& m )
{
    const uintmax_t UINTMAX_T_MAX = ~((uintmax_t)0);
    uintmax_t mm = static_cast<uintmax_t>(n - m) + 1; // mm is now the lower
range
    uintmax_t nn = static_cast<uintmax_t>(n);
    uintmax_t kk, prod;
    int_precision res = int_precision(1);

    if (n.size() > 1 || m > n ) // more >= 2^64! or m > n
        throw int_precision::domain_error();

    if (nn <= 1)
        return res;
    for (kk = 1; nn > mm; --nn, ++mm)
    {
        prod = nn * mm; // Could potentially overflow
        // if no overflow on both conditions, then safely multiply it to kk
        if (prod < nn || kk < UINTMAX_T_MAX / prod)
            kk *= prod;
        else
        { // multiply the current kk to the result
            res *= int_precision(kk);
            if(prod < nn)
            { // overflow in pairwise multiplication
                res *= int_precision(nn);
                prod = mm;
            }
            kk = prod;
        }
    }
    res *= int_precision(kk);
    if (nn == mm)
```

Fast Factorial & Binomial Computation for Arbitrary Precision

```
        res *= int_precision(nn);
    return res;
}
```

Now that we have the falling factorial, we can improve the factorial by splitting a factorial call for $n!$ into two calls.

$$\frac{n!}{(n-m)!} \text{ and } (n-m)! \quad (19)$$

Where we can run each calculation in parallel, see the source below, which uses C++ threading.

There is some overhead in creating a thread, so it's usually not worthwhile to use threading for falling factorial values exceeding 2000!

A more interesting method is the binary splitting method for the falling factorial.

Falling Factorial using binary splitting

Like the factorial, we can gain some speed by reusing the binary splitting method for the falling factorial by noticing that. n^m Indeed, the two parameters of the binary splitting method are n and m , and, unsurprisingly, we observe the same performance gain as with the factorial method, with the crossover occurring around 2,000, at which point the binary splitting method outperforms the factorial method.

A threaded version of the binary splitting method first outperformed the standard binary splitting method by approximately 5,000. We present the source code for our final hybrid version, based on the notes above. n and m are the n^m From the definition of the falling factorials.

Source falling factorial hybrid implementation.

```
// Falling factorial as a hybrid thread version
// notice that maximum n & m is limited to 2^64-1. we should be enough for most
// need
int_precision fallingfactorial_hybrid_thread(const int_precision& n, const
int_precision& m )
{
    const int_precision c0(0), c1(1), c2(2);

    auto fallingfactorial_loop = [&](const int_precision& n, const
int_precision& m)
    {
        const uintmax_t UINTMAX_T_MAX = ~(0ull);
        uintmax_t mm = static_cast<uintmax_t>(n - m)+1; // mm is now the
lower range
        uintmax_t nn = static_cast<uintmax_t>(n), kk;
        int_precision res = c1;

        if (nn <= 1)
            return res;
        for (kk = 1; nn > mm; --nn, ++mm)
        {
            uintmax_t prod;
            prod = nn * mm; // Never overflow since n <= 2'000
            if (kk < UINTMAX_T_MAX / prod)
```

Fast Factorial & Binomial Computation for Arbitrary Precision

```
        kk *= prod;
    }
    else
    {
        res *= int_precision(kk);
        kk = prod;
    }
}
res *= int_precision(kk);
if (nn == mm)
    res *= int_precision(nn);
return res;
};

// We need to use the std::function to be able to call the
// lambda function recursively.
std::function<int_precision(const int_precision&, const int_precision&)>
fallingfactorial_binary_splitting = [&](const int_precision& a, const
int_precision& b)
{
    const int_precision diff = a - b;

    switch (static_cast<uintmax_t>(diff))
    {
        // Base cases
        case 0: return c1; break;
        case 1: return a; break;
        case 2: return a * (a - c1); break;
        case 3: return a * (a - c1) * (a - c2); break;
        default:// Fall through
            break;
    }
    int_precision m = (a + b) / c2;
    return fallingfactorial_binary_splitting(a, m) *
fallingfactorial_binary_splitting(m, b);
};

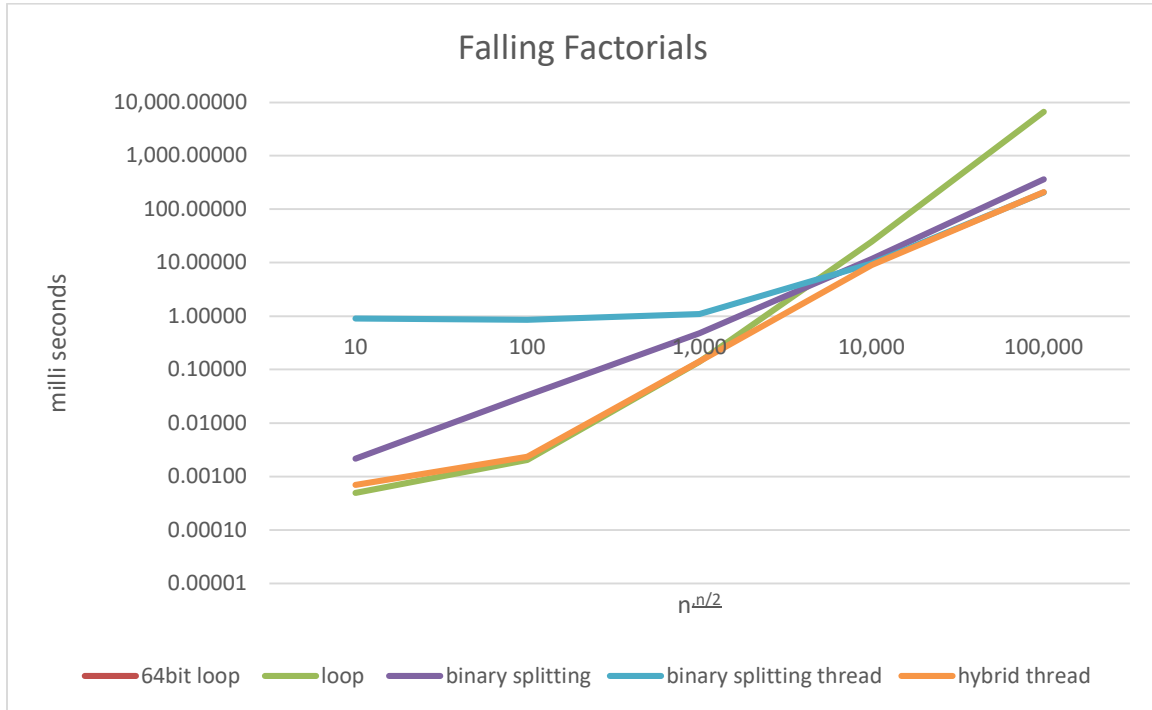
// The balance loop base is fastest below 2000!
if (n <= int_precision(2000))
    return fallingfactorial_loop(n,m);

// Do the binary splitting method above 5000!
// with or without threading. 2 ways Threading improves the speed with
// 30-60% over the non-threaded version
int_precision mm = n - m, mid = (n + mm) / c2, high, low;
if (n <= int_precision(5000))
{
    high = fallingfactorial_binary_splitting(n, mid);
    low = fallingfactorial_binary_splitting(mid, mm);
}
else
{
    std::thread first([&]()
    {
        high = fallingfactorial_binary_splitting(n, mid);
    });
    // interval [n...mid]
    std::thread second([&]()
    {
        low = fallingfactorial_binary_splitting(mid, mm);
    });
    // interval [mid...mm]
    first.join();
    second.join();
}
}
```

Fast Factorial & Binomial Computation for Arbitrary Precision

```
} return high * low;
```

Performance of falling factorial



We observe a pattern similar to the factorial case: the loop-based approach is faster up to approximately 5,000 factorizations. Then the binary splitting method takes over. Enabling threading also provides a performance boost of a similar magnitude. The hybrid thread combines the loop-based falling factorial and the binary splitting method.

Rising Factorial

The rising factorial describes the factorial between two numbers m and n in rising (or ascending) notation:

$$n^{\overline{m}} = n \cdot (n + 1) \cdot (n + 2) \dots (n + m - 1) = \prod_{k=n}^{n+m-1} k \quad (20)$$

However, instead of creating a nearly similar function as for the falling factorial, you can map the rising factorial into the falling factorial and reuse some code.

$$n^{\overline{m}} = (n + m - 1)^{\underline{m}} \quad (21)$$

Binomial coefficients

Fast Factorial & Binomial Computation for Arbitrary Precision

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \text{ where } 0 \leq m \leq n \quad (22)$$

As noted under partial factorials, the binomial coefficients can be computed using two different factorial functions.

Source for binomial

```
// binomial computation using arbitrary precision
int_precision binomial(const int_precision& n, const int_precision& m)
{
    int_precision nn(n), mm(m);
    if (mm > nn) return int_precision(0); // Base case
    if (mm > nn - mm) // Use the identity to reduce m
        mm = nn - mm;
    return fallingfactorial(nn, mm)/factorial(mm);
}
```

Notice that the binomial implementation avoids intermediate overflow by using falling factorials rather than naive factorial division. In the above function, we also use the identity that:

$$\binom{n}{m} = \binom{n}{n-m} \quad (23)$$

Sometimes it is necessary to calculate a sequence of binomial coefficients. For example, calculating Bernoulli's number requires it. This can lead to optimization if you instead use one of the two binomial recurrences:

$$\binom{n+1}{m} = \frac{n+1}{n-m+1} \binom{n}{m} = \binom{n}{m} + \binom{n}{m-1} \quad (24)$$

Or:

$$\binom{n}{m+1} = \frac{n-m}{m+1} \binom{n}{m} \quad (25)$$

Reference

- 1) Arbitrary precision library package. [Arbitrary Precision C++ Packages \(hvks.com\)](http://hvks.com)
- 2) Numerical recipes in C++, 3rd edition, Cambridge University Press, New York, NY 2007
- 3) Henrik Vestermark, "HVE The math behind arbitrary". [HVE The Math behind arbitrary precision.docx \(hvks.com\)](http://hvks.com)