Abstract

There are several formulas available to compute sums of powers of natural numbers. Although the implementation of these formulas is straightforward on a computer, computing the exact sums efficiently requires the use of multi-precision arithmetic and proper tuning of the code. This paper does a theoretical as well as practical analysis of efficient implementations of different formulas and compares their performances. It concludes with a note that the best formula is a lesser-known one.

PowerSum: Efficient Implementations of Power Sum Computation

Mariappan Asokan

June 20, 2025

1 Introduction

Finding the sums of integer powers of natural numbers

$$S_m(n) = \sum_{i=1}^{n} i^m = 1^m + 2^m + \ldots + n^m$$

is a well studied problem for centuries. Mathematicians like Johann Faulhaber and Jakob Bernoulli came up with closed form solutions for the sums of power problem. They are faster to compute since the number of terms to compute depends on m not on n and generally $m < n$. Different solutions can be derived from identities in algebra or combinatorics resulting in different formulas.

Although these formulas are straightforward to program on a computer, there are a couple of problems:

- For higher values of m and n, computation of $S_m(n)$ may cause overflow of integers in many programming languages. For example,

  $$S_9(107) = 20603628525008761668$$

  This is larger than the maximum integer value supported.

- Some formulas have fractional terms that need to be reduced to get the final integral value. Fractional arithmetic without losing precision is not supported in many programming languages.

To alleviate these problems, we can make use of libraries such as `gmp` that support arbitrary precision arithmetic on integers and rational numbers. The goal of this paper is to compare arbitrary precision implementations of these formulas with respect to the following criteria:

- faster execution time
- less code complexity

Rest of the paper is arranged as follows. First, a brief overview of arbitrary precision arithmetic is provided. It will be followed by a discussion on a simple implementation without using any formula. Next a set of known formulas, an analysis of each of their implementations, and the results of running tests are presented. At the end, test results are interpreted for the best formula. The code is made available as open source software in the following programming languages: C, C++, Java, and Python.

2 Arbitrary precision arithmetic

The builtin integers in many programming languages have a maximum limit on values stored in them. For example, in C, an `unsigned long int` can hold a maximum value of `ULONG_MAX` which is defined in `limits.h`. For a 64-bit system, it will be 18446744073709551615. The arbitrary precision arithmetic libraries use an array of integers (called limbs) to store values larger than this. Arithmetic operation on the array is performed using techniques taught in elementary school. For example, addition of two integers will be done by adding one limb at a time from right to left and propagating the carry to the next limb. Multiplication and division use somewhat more sophisticated algorithms to reduce the CPU time. For more details on different algorithms, the reader can see online documentation available at [3] for the GNU Multi-precision Library. This library is widely used in C, C++, and Python. Java has `BigInteger` class available in the `math` package. The source code for `BigInteger` has extensive documentation on the algorithms used.

To summarize:

- For an arbitrary precision integer $N, N_b = O(\log_2 N)$ bits are needed for storage.
- Integer addition, subtraction, and bit operations take $O(N_b)$ time.
- Integer multiplication takes $O(N_b^k)$ time where $1.404 \leq k \leq 2$.
- Asymptotically, integer division takes the same time as multiplication but differs by a constant factor.
- Arithmetic operations on rational numbers trigger multiple integer multiplication and division operations. Reduction of a fraction involves computing GCD of numerator and denominator. This triggers even more integer multiplication and division operations.

3 A simple implementation

$S_m(n)$ can be computed by simply adding the powers of integers up to n inclusive. The code for this has the lowest logical complexity and uses the least memory compared to the formula implementations. Of course, the execution time is not the best. This will be used as a basis to compare with the implementation of the formulas both in terms of theoretical time complexity as well as the real CPU time to execute the code.

3.1 Analysis of the simple implementation

Lemma 1. *The computation of N^m can be done in $O(m^k \log_2^k N)$ time where $1.404 \leq k \leq 2$.*

Proof. N^m can be computed by repeated squaring of N. There will be $O(\log_2 m)$ squaring operations. Each squaring operation doubles the number of bits. So, the costs of squaring operations successively are:

$O(N_b^k), O((2N_b)^k), O((4N_b)^k), \ldots, O((2^{\log_2 m} N_b)^k)$.

where $1.404 \leq k \leq 2$ is the time complexity for multiplication.

The total cost can be computed as follows (O notation is omitted where it is obvious):

$$\begin{aligned}
\sum_{i=0}^{\log_2 m} (2^i N_b)^k &= N_b^k \sum_{i=0}^{\log_2 m} (2^i)^k \\
&= N_b^k (2^0 + 2^1 + \ldots + 2^{\log_2 m})^k \\
&= N_b^k \left(\frac{2^{\log_2 m + 1} - 1}{2 - 1} \right)^k \\
&= O(N_b^k (2^{\log_2 m})^k) \\
&= O(N_b^k (m)^k) \\
&= O(m^k N_b^k) \\
&= O(m^k \log_2^k N)
\end{aligned}$$

□

Theorem 1. *$S_m(n)$ can be computed with simple implementation in $O(nm^k \log_2^k n)$ time where $1.404 \leq k \leq 2$.*

Proof. Out of n integers, $\frac{n}{2}$ of them have $\lceil \log_2 n \rceil$ bits, $\frac{n}{4}$ of them have $\lceil \log_2 n \rceil - 1$ bits, and so on. To compute $S_m(n)$,

$$\begin{aligned}
\text{total time} &= \frac{n}{2} O(m^k \log_2^k n) + \frac{n}{4} O(m^k (\log_2 n - 1)^k) + \ldots \\
&= O(nm^k \log_2^k n)
\end{aligned}$$

where $1.404 \leq k \leq 2$. Here we used Lemma 1 for the cost of computing the m^{th} power of an integer. □

4 Known formulas

In this section, a set of known formulas for computing $S_m(n)$ are presented. Without loss of generality, some formulas support $m = 0$ and $n = 0$ as well. The case of $m = 0$ is special:

$$S_0(0) = 0^0 + 1^0 + 2^0 + \ldots + n^0 = n + 1$$

Note that $0^0 = 1$. For $m > 0, 0^m = 0$ and so we get the sum for the powers of natural numbers.

4.1 Faulhaber's formula

Faulhaber found out that for odd m, $S_m(n)$ can be expressed as a function of the triangular number $T = \frac{n(n+1)}{2}$ which itself is $S_1(n)$. More precisely,

$$S_{2m+1}(n) = \sum_{j=1}^{m+1} F_{m,j} T^j \tag{1}$$

where $F_{m,j}$ are coefficients that Faulhaber himself came up with for odd powers up to 17 [4]. Some Faulhaber coefficients are shown in table 1.

Table 1: Faulhaber coefficients

	j								
m	1	2	3	4	5	6	7	8	9
0	1								
1	0	1							
2	0	$-\frac{1}{3}$	$\frac{4}{3}$						
3	0	$\frac{2}{6}$	$-\frac{8}{6}$	$\frac{12}{6}$					
4	0	$-\frac{3}{5}$	$\frac{12}{5}$	$-\frac{20}{5}$	$\frac{16}{5}$				
5	0	$\frac{5}{6}$	$-\frac{40}{6}$	$\frac{68}{6}$	$-\frac{64}{6}$	$\frac{32}{6}$			
6	0	$-\frac{691}{105}$	$\frac{2764}{105}$	$-\frac{4720}{105}$	$\frac{4592}{105}$	$-\frac{2800}{105}$	$\frac{960}{105}$		
7	0	$\frac{420}{12}$	$-\frac{1680}{12}$	$\frac{2872}{12}$	$-\frac{2816}{12}$	$\frac{1792}{12}$	$-\frac{768}{12}$	$\frac{192}{12}$	
8	0	$-\frac{10851}{45}$	$\frac{43404}{45}$	$-\frac{74220}{45}$	$\frac{72912}{45}$	$-\frac{46880}{45}$	$\frac{21120}{45}$	$-\frac{6720}{45}$	$\frac{1280}{45}$

There is no simple relationship among the coefficients in the formula. A. W. F. Edwards [1] showed that the coefficients can be obtained by matrix inversion. He also showed that similar formulas exist for even powers. According to him:

$$\begin{aligned} S_{2m+1}(n) &= \frac{1}{2} \sum_{j=1}^{m+1} f_{m,j} N^j \\ S_{2m}(n) &= \frac{(2n+1)}{2} \sum_{j=1}^{m+1} g_{m,j} N^j \end{aligned} \tag{2}$$

where $N = n(n+1)$. This is slightly different from (1). The coefficients $f_{m,j}$ and $g_{m,j}$ can be computed by a matrix inversion. The coefficients $F_{m,j}$ and $f_{m,j}$ differ by constant factors which are powers of 2.

Without loss of generality, the ensuing discussion assumes odd m. It is applicable for even m as well. For odd powers, Edwards starts with the identity

$$[n(n+1)]^m = 2\left[\binom{m}{1}S_{2m-1}(n) + \binom{m}{3}S_{2m-3}(n) + \ldots + \binom{m}{2m-1}S_1(n)\right] \tag{3}$$

It is assumed that for $k > m$, $\binom{m}{k} = 0$.
For $m = 2, 3, 4, \ldots$ he expresses (3) in matrix form as

$$\begin{bmatrix} N^2 \\ N^3 \\ N^4 \\ N^5 \\ \vdots \end{bmatrix} = 2 \begin{bmatrix} 2 & 0 & 0 & 0 & \cdots \\ 1 & 3 & 0 & 0 & \cdots \\ 0 & 4 & 4 & 0 & \cdots \\ 0 & 1 & 10 & 5 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} S_3(n) \\ S_5(n) \\ S_7(n) \\ S_9(n) \\ \vdots \end{bmatrix}$$

He then takes the inverse of the matrix to solve for desired $S_{2m-1}(n)$ as below:

$$\begin{bmatrix} S_3(n) \\ S_5(n) \\ S_7(n) \\ S_9(n) \\ \vdots \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 2 & 0 & 0 & 0 & \cdots \\ 1 & 3 & 0 & 0 & \cdots \\ 0 & 4 & 4 & 0 & \cdots \\ 0 & 1 & 10 & 5 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}^{-1} \begin{bmatrix} N^2 \\ N^3 \\ N^4 \\ N^5 \\ \vdots \end{bmatrix}$$

Here $N = n(n+1)$ is used for brevity. Since we are interested in computing $f_{m,j}$ and $g_{m,j}$ for a particular m, we will show that there is no need to keep the full matrix in memory nor to compute the full inverse of the matrix.

First, the original matrix can be rewritten as

$$\begin{bmatrix} N^m \\ N^{m-1} \\ \vdots \\ N^5 \\ N^4 \\ N^3 \\ N^2 \end{bmatrix} = 2 \begin{bmatrix} \binom{m}{1} & \binom{m}{3} & \binom{m}{5} & \cdots & \cdots & \cdots & \cdots \\ 0 & \binom{m-1}{1} & \binom{m-1}{3} & \binom{m-1}{5} & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 5 & 10 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 4 & 4 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 2 \end{bmatrix} \begin{bmatrix} S_{2m-1}(n) \\ S_{2m-3}(n) \\ \vdots \\ S_9(n) \\ S_7(n) \\ S_5(n) \\ S_3(n) \end{bmatrix} \tag{4}$$

We note that the matrix is in row echelon form. Secondly, the column vector of S can be treated as a solution to a system of linear equations solvable by Gauss-Jordan elimination. Since the matrix is already in row echelon form, it is enough to convert it to reduced row echelon form. There is no need to generate the entire matrix at once since we are interested in only $S_{2m-1}(n)$ for a particular value of m. We can generate the first row and keep it in memory. Subsequent rows can be generated one at a time. The reduction of the first row can be done by pivoting on the columns one at a time. A run through an example will clarify the steps. Table 2 shows the steps in deriving coefficients to compute $S_9(n)$. The first row is shown in bold letters for easy visualization of the transformations it goes through as it is reduced with other rows one at a time.

Table 2: Steps involved in computing $S_9(n)$

Reduction process					Comment
5	**10**	**1**	**0**	$\boldsymbol{N^5}$	Augmented row1
1	**2**	$\boldsymbol{\frac{1}{5}}$	**0**	$\boldsymbol{\frac{N^5}{5}}$	row1 ← row1/col1
0	4	4	0	N^4	Augmented row2
0	1	1	0	$\frac{N^4}{4}$	row2 ← row2/col2
1	**0**	$\boldsymbol{-\frac{9}{5}}$	**0**	$\boldsymbol{(\frac{N^5}{5} - \frac{N^4}{2})}$	row1 ← row1 - 2*row2
0	0	3	1	N^3	Augmented row3 - Note: row2 is not needed anymore
0	0	1	$\frac{1}{3}$	$\frac{N^3}{3}$	row3 ← row3/col3
1	**0**	**0**	$\boldsymbol{\frac{3}{5}}$	$\boldsymbol{(\frac{N^5}{5} - \frac{N^4}{2} + \frac{3N^3}{5})}$	row1 ← row1 + $(\frac{9}{5})$*row3
0	0	0	2	N^2	Augmented row4 - Note: row3 is not needed anymore
0	0	0	1	$\frac{N^2}{2}$	row4 ← row4/col4
1	**0**	**0**	**0**	$\boldsymbol{(\frac{N^5}{5} - \frac{N^4}{2} + \frac{3N^3}{5} - \frac{3N^2}{10})}$	row1 ← row1 - $(\frac{3}{5})$*row4

$\therefore S_9(n) = \frac{1}{2}\left(\frac{N^5}{5} - \frac{N^4}{2} + \frac{3N^3}{5} - \frac{3N^2}{10}\right).$

4.1.1 Implementation notes

While generating a row, each binomial coefficient except the first one can be computed from the previous one as:

$$\binom{m}{k+2} = \binom{m}{k}\left[\frac{(m-k)(m-k-1)}{(k+1)(k+2)}\right]$$

The first one takes constant time since it is trivial. This implies that all the binomial coefficients in RHS of (3) can be computed in $O(m)$ steps. Coming up with each coefficient for $S_m(n)$ requires $O(m)$ steps to compute a row of the matrix plus $O(m)$ rational number additions or subtractions. To compute all the coefficients, it takes $O(m^2)$ steps. Overall, the space complexity is $O(m)$. Unlike the values kept in rational numbers as shown in table 2, the values in each row can be kept as integers. The scaling can be postponed till the end when a new coefficient is generated. This will avoid numerous arithmetic operations in rational numbers which are more expensive.

Obviously, each N^j can be computed from N^{j-1} in constant time. So $S_m(n)$ can be computed in $O(m)$ steps once the coefficients are computed.

4.2 Formula with Bernoulli numbers as coefficients

This is the most popular formula that one can find in many combinatorics text books with a proof based on induction. For example, see chapter 6 in "Concrete Mathematics" [2]. This formula is defined as:

$$S_m(n) = \frac{1}{m+1} \sum_{j=0}^{m} \binom{m+1}{j} B_j (n+1)^{m+1-j} \tag{5}$$

where B_j are Bernoulli numbers as defined by the recurrence relation:

$$\begin{aligned} B_0 &= 1 \\ \sum_{j=0}^{k} \binom{k+1}{j} B_j &= 0 \end{aligned} \tag{6}$$

Some Bernoulli numbers are listed in table 3. Readers may notice that for $m > 1, B_m = 0$ when m is odd.

Table 3: Some Bernoulli numbers

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B_k	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$	0	$-\frac{1}{30}$	0	$\frac{5}{66}$	0	$-\frac{691}{2730}$	0	$\frac{7}{6}$

4.2.1 Implementation notes

The Bernoulli number B_j is zero when $j > 1$ and odd. This means that only half the terms have to be computed for $S_m(n)$. In (6), the first coefficient is always 1. Each other $\binom{k+1}{j}$ can be computed from its previous one in constant time as

$$\binom{k+1}{j} = \binom{k+1}{j-1} \left[\frac{k+2-j}{j}\right]$$

To compute $S_m(n)$, the highest coefficient needed is B_m. Each B_j can be computed in at most $O(m)$ steps. Each step involves an addition of rational numbers. All B_js can be computed in $O(m^2)$ steps. To store all the coefficients, $O(m)$ space is needed. The binomial coefficients in each term of (6) can be computed from the previous one with any multiplication or division done as integers before multiplying by fractional B_js.

While computing $S_m(n)$ using (5), the terms can be added from right to left (that is starting $j = m$ and decrementing j until $j = 0$ inclusive) since the power of $(n + 1)$ increases in that direction. This means we can start with the lowest power and multiply by $(n+1)$ each time to get the next power. Had we started from left to right, we would have computed the highest power first and divided by $(n + 1)$ to get the next lower power. Since division is more expensive than multiplication, we opt to add from right to left. Also, the integer components in each term namely the power of $(n+1)$ and the binomial coefficient can be computed first before multiplying by B_j. This is done to reduce the number of rational arithmetic operations. The binomial coefficient in each term can be computed from the previous one in constant time similar to when computing B_js. The binomial coefficient in the first term is 1 which requires constant time to compute. Thus computing $S_m(n)$ takes $O(m)$ steps once the coefficients are computed.

4.3 Formulas with Stirling number of second kind as coefficients

There are two different formulas. The first formula can be derived from the following combinatorial identity that readers can find in chapter 6 in [2]:

$$x^m = \sum_{k=0}^{m} \left\{ {m \atop k} \right\} x^{\underline{k}}$$

Here the coefficient $\left\{ {m \atop k} \right\}$ is Stirling number of second kind which is recursively defined as follows:

$$\begin{aligned} \left\{ {m \atop k} \right\} &= m \left\{ {m \atop k-1} \right\} + \left\{ {m-1 \atop k-1} \right\} \\ \left\{ {0 \atop 0} \right\} &= 1 \\ \left\{ {m \atop 0} \right\} &= 0 \quad \forall m > 0 \\ \left\{ {m \atop k} \right\} &= 0 \quad \forall k > m \end{aligned}$$

For $k > 0, x^{\underline{k}} = x(x-1)(x-2)\ldots(x-k+1)$ is the falling factorial of x.

$S_m(n)$ can be computed as

$$\begin{aligned}
S_m(n) &= \sum_{x=0}^{n} x^m \\
&= \sum_{x=0}^{n} \sum_{k=0}^{m} {m \brace k} x^{\underline{k}} \\
&= \sum_{k=0}^{m} {m \brace k} \sum_{x=0}^{n} x^{\underline{k}} \\
&= \sum_{k=0}^{m} {m \brace k} k! \sum_{x=0}^{n} \binom{x}{k} \qquad (7)\\
&= \sum_{k=0}^{m} {m \brace k} k! \binom{n+1}{k+1} \leftarrow \text{used Hockey stick identity [7]} \\
&= \sum_{k=0}^{m} {m \brace k} k! \frac{(n+1)^{\underline{k+1}}}{(k+1)!} \\
&= \sum_{k=0}^{m} {m \brace k} \frac{(n+1)^{\underline{k+1}}}{(k+1)}
\end{aligned}$$

There is another combinatorial identity involving Stirling numbers of second kind that can be found in chapter 6 in [2] as well.

$$x^m = \sum_{k=0}^{m} {m \brace k} (-1)^{m-k} x^{\overline{k}}$$

where $x^{\overline{k}} = x(x+1)\dots(x+k-1)$ is the rising factorial. Following similar steps as for (7), we can derive the other formula for $S_m(n)$ using this identity as

$$S_m(n) = \sum_{j=0}^{m} {m \brace j} (-1)^{m-j} \frac{(n+j)^{\underline{j+1}}}{(j+1)} \qquad (8)$$

Some Stirling numbers of second kind are listed in table 4.

4.3.1 Implementation notes

Out of the two variants, formula (7) is preferable since $S_m(n)$ contains only positive terms. It will make convergence towards final sum monotonically as we add terms. (8) may generate intermediate sums that are larger which cost more to work with. Unlike Bernoulli and Faulhaber coefficients, all Stirling numbers of second kind are integers. To compute $S_m(n)$, we need to generate m coefficients. When done iteratively starting from $m = 0$, this will take $O(m^2)$ steps. Each step involves an integer multiplication and addition. In table 4 values in one row can be computed only from previous row thus requiring $O(m)$ space.

Table 4: Stirling numbers of second kind

	j										
m	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	0	1									
2	0	1	1								
3	0	1	3	1							
4	0	1	7	6	1						
5	0	1	15	25	10	1					
6	0	1	31	90	65	15	1				
7	0	1	63	301	350	140	21	1			
8	0	1	127	966	1701	1050	266	28	1		
9	0	1	255	3025	7770	6951	2646	462	36	1	
10	0	1	511	9330	34105	42525	22827	5880	750	45	1

When $n < m$, there is no need to compute all the terms as the falling factorial will be 0. To be precise, it is sufficient to initialize $\min(m, n) + 1$ coefficients and compute that many terms.

The falling factorial in each term in the formula for $S_m(n)$ can be computed from the previous one in constant time as:

$$n^{\underline{k+1}} = n^{\underline{k}}(n-k)$$

The first falling factorial is $n+1$ and it takes constant time to compute it. This means computing $S_m(n)$ takes $O(m)$ steps once the coefficients are computed.

4.4 Formula with Euler numbers of first kind as coefficients

This formula can be derived from the following combinatorial identity that readers can find in chapter 6 in [2] ("Worpitzky's identity"):

$$x^m = \sum_{k=0}^{m} \left\langle {m \atop k} \right\rangle \binom{x+k}{m}$$

where $m \geq 0$ and $\left\langle {m \atop k} \right\rangle$ is Euler number of first kind defined recursively as:

$$\left\langle {i \atop 0} \right\rangle = 1 \quad \forall i >= 0$$
$$\left\langle {i \atop k} \right\rangle = (k+1)\left\langle {i-1 \atop k} \right\rangle + (i-k)\left\langle {i-1 \atop k-1} \right\rangle$$

$S_m(n)$ can be computed as

$$\begin{aligned} S_m(n) &= \sum_{x=0}^{n} x^m \\ &= \sum_{x=0}^{n}\sum_{k=0}^{m} \left\langle {m \atop k} \right\rangle \binom{x+k}{m} \\ &= \sum_{k=0}^{m} \left\langle {m \atop k} \right\rangle \sum_{x=0}^{n} \binom{x+k}{m} \\ &= \sum_{k=0}^{m} \left\langle {m \atop k} \right\rangle \binom{n+k+1}{m+1} \leftarrow \text{used Hockey stick identity [7]} \end{aligned}$$

Some Euler numbers of first kind are shown in table 5. Readers will notice that half the number of values are mirror reflections of others around the center.

Table 5: Euler numbers of first kind

	j										
i	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	1	0									
2	1	1	0								
3	1	4	1	0							
4	1	11	11	1	0						
5	1	26	66	26	1	0					
6	1	57	302	302	57	1	0				
7	1	120	1191	2416	1191	120	1	0			
8	1	247	4293	15619	15619	4293	247	1	0		
9	1	502	14608	88234	156190	88234	14608	502	1	0	
10	1	1013	47840	455192	1310354	1310354	455192	47840	1013	1	0

4.4.1 Implementation notes

Like Stirling numbers of second kind, Euler numbers of first kind are all integers. Also, the number of steps needed to initialize all the coefficients in $S_m(n)$ is $O(m^2)$. Each step involves two integer multiplications and one integer addition. Euler numbers of first kind are symmetrical. We need to initialize only half of them. Others are just mirror reflections. It is sufficient to compute $\min(m, n) + 1$ terms starting from the right. All other leading terms evaluate to 0.

In the formula for $S_m(n)$, the binomial coefficient in each term can be computed from the previous one in constant time as:

$$\binom{n+1}{m+1} = \binom{n}{m+1}\left[\frac{n+1}{n-m}\right]$$

The first binomial coefficient can be computed in $O(m)$ steps. So, computing $S_m(n)$ requires $O(m)$ steps once the coefficients are computed.

4.5 Formula with central factorial numbers of second kind as coefficients

[4] refers to the following combinatorial identities from [5]:

$$\begin{aligned} x^{2m-1} &= \sum_{k=1}^{m}(2k-1)!T(2m,2k)\binom{x+k-1}{2k-1} \\ x^{2m} &= \sum_{k=1}^{m}\frac{(2k)!}{2}T(2m,2k)\left[\binom{x+k}{2k}+\binom{x+k-1}{2k}\right] \end{aligned} \qquad (9)$$

where $T(2m,2m)$ known as central factorial number of second kind is defined recursively as:

$$\begin{aligned} T(2m,2m) &= 1 \\ T(2m,2k) &= T(2m-2,2k-2)+k^2T(2m-2,2k) \end{aligned}$$

The sum formulas for odd and even m can be derived from (9) using Hockey stick identity [7] as:

$$\begin{aligned}
S_{2m-1}(n) &= \sum_{k=1}^{m}(2k-1)!T(2m,2k)\binom{n+k}{2k} \\
&= \sum_{k=1}^{m}(2k-1)!T(2m,2k)\frac{(n+k)!}{(n-k)!(2k)!} \\
&= \sum_{k=1}^{m}T(2m,2k)\frac{(n+k)^{\underline{2k}}}{2k} \\
S_{2m}(n) &= \sum_{k=1}^{m}\frac{(2k)!}{2}T(2m,2k)\left[\binom{n+k+1}{2k+1}+\binom{n+k}{2k+1}\right] \\
&= \sum_{k=1}^{m}\frac{(2k)!}{2}T(2m,2k)\left[\frac{(n+k+1)!}{(n-k)!(2k+1)!}+\frac{(n+k)!}{(n-k-1)!(2k+1)!}\right] \\
&= \sum_{k=1}^{m}\frac{(2k)!}{2}\frac{T(2m,2k)}{(2k+1)!}\left[\frac{(n+k+1)!}{(n-k)!}+\frac{(n+k)!}{(n-k-1)!}\right] \\
&= \sum_{k=1}^{m}\frac{T(2m,2k)}{2(2k+1)}\left[\frac{(n+k)!(n+k+1)}{(n-k)!}+\frac{(n+k)!(n-k)}{(n-k)!}\right] \\
&= \sum_{k=1}^{m}\frac{T(2m,2k)}{2(2k+1)}\frac{(n+k)!}{(n-k)!}(n+k+1+n-k) \\
&= \sum_{k=1}^{m}T(2m,2k)\frac{2n+1}{2(2k+1)}\frac{(n+k)!}{(n-k)!} \\
&= \sum_{k=1}^{m}T(2m,2k)\frac{2n+1}{2(2k+1)}(n+k)^{\underline{2k}}
\end{aligned}$$

Table 6 lists some Central factorial numbers of second kind.

Table 6: Central factorial numbers of second kind

	$2k$					
$2m$	0	2	4	6	8	10
0	1					
2	0	1				
4	0	1	1			
6	0	1	5	1		
8	0	1	21	14	1	
10	0	1	85	147	30	1

4.5.1 Implementation notes

The central factorial numbers of second kind are all integers. To compute $S_m(n)$, we need only $\lceil \frac{m}{2} \rceil$ coefficients. The computation of these coefficients takes $O(m^2)$ steps and $O(m)$ space. Each step involves two integer multiplications and one integer addition. This formula has only $\lceil \frac{m}{2} \rceil$ number of coefficients. It is sufficient to initialize $\min(\lceil \frac{m}{2} \rceil, n)$ coefficients and compute that many terms.

Once the coefficients are computed, computing $S_m(n)$ requires $O(m)$ steps since the falling factorial in each term can be computed from the previous one in constant time as:

$$(n+k+1)^{\underline{2(k+1)}} = (n+k)^{\underline{2k}}(n+k+1)(n-k+2)$$

The first falling factorial value is 1 and it takes constant time to compute.

4.6 Analysis of the formulas

Some common characteristics we observe in all the formulas are:

- they consist of a polynomial in n with $O(m)$ terms to compute
- each term has a coefficient to compute first

For each formula, we know that there are $O(m^2)$ steps to compute all the coefficients. In order to compute the cost of each step, we need to know how big the coefficients are since the cost of multi-precision arithmetic depends on that. We will compute some upper bounds on the values of the coefficients. In reality, the values will be much lower than the upper bound due to the complex nature of the coefficient computation. However, for the sake of our analysis we use the upper bound.

Lemma 2. *The value of highest coefficient in the formulas for $S_m(n)$ is in the order of m^m.*

Proof. In Faulhaber formula (4), there are $\lceil \frac{m}{2} \rceil$ binomial coefficients in the first row. The maximum value of the binomial coefficient is in the order of $2^{\frac{m}{2}}$. The first row gets multiplied by the pivot column in each subsequent row on each iteration to create a new coefficient. The pivot column values successively are $\binom{m-1}{1}$, $\binom{m-2}{1}$, and so on. Since there are $\lceil \frac{m}{2} \rceil$ rows, the maximum value attained by a coefficient is in the order of $2^{\frac{m}{2}} m^{\frac{m}{2}}$ or m^m. Note that we are ignoring the fact that this value gets scaled each time to generate a new coefficient. Sometimes, the fraction may get reduced and sometimes it may not. When it gets reduced, the numerator in the final coefficient value may be much lower than this maximum.

In Bernoulli formula, $B_k = 0$ for odd $k > 1$. For even k, according to [6] the closed form of a Bernoulli number can be computed using

$$B_k = (-1)^{\frac{k}{2}+1} \frac{2\zeta(k)k!}{(2\pi)^k}$$

where $\zeta(k)$ is the Riemann zeta function. Also, for large k,

$$|B_k| \sim 4\sqrt{\frac{\pi k}{2}}\left(\frac{k}{2\pi e}\right)^k$$

This implies that the coefficients in Bernoulli formula can be in the order of m^m.

In Stirling and Euler's formulas, at each iteration a coefficient value from the previous iteration gets multiplied by a quantity which is in the order of m. Since there are m iterations, a coefficient can be in the order of m^m.

In Central factorial formula, at each iteration a coefficient value from the previous iteration gets multiplied by a quantity which is in the order of m^2. Since there are $\lceil \frac{m}{2} \rceil$ iterations, the maximum value of a coefficient can be in the order of m^m. $\square$

In the following theorem we prove the asymptotic time complexity for the computation of the coefficients in the formulas.

Theorem 2. *In the formulas for $S_m(n)$, the coefficients can be computed in $O(m^{k+2}\log_2^k m)$ time where $1.404 \leq k \leq 2$.*

Proof. While computing the coefficients, each step in the formulas requires a constant number of integer multiplications, divisions, and additions. The costliest is either multiplication (for all formulas except Bernoulli) or division (for Bernoulli formula.) By Lemma 2, the coefficient values can grow to be in the order of m^m. So the cost of each step is $O((\log_2 m^m)^k)$ or $O(m^k \log_2^k m)$ where $1.404 \leq k \leq 2$. Since there are $O(m^2)$ steps, the total cost is $O(m^{k+2}\log_2^k m)$. $\square$

Assuming that the coefficients in the formulas are available, we now proceed to analyze the time complexity of computing the polynomial in n.

Theorem 3. *For all formulas once the coefficients are known, $S_m(n)$ can be computed in $O(m^{k+1}\log_2^k mn)$ time where $1.404 \leq k \leq 2$.*

Proof. Once the coefficients in the formulas are known, the computation of each term involves a fixed number of integer multiplications and divisions on the previous term. Each term consists of a coefficient and polynomial in n as factors. The polynomial portion each can be evaluated from the previous one in constant time. The first one takes at most $O(m)$ steps. Since there are $O(m)$ terms in the formula, overall it takes $O(m)$ steps to compute the sum. Each term can grow as big as $m^m n^m$ in value since the coefficient can grow as big as m^m and the terms may grow as big as n^m. Each step involves a multiplication and division. So it takes $O(m^k \log_2^k mn)$ time where $1.404 \leq k \leq 2$. The time to compute all the terms is $O(m^{k+1}\log_2^k mn)$. $\square$

In all cases, the coefficient initialization time is larger than the time to compute all the terms in the formulas. According to Theorem 1 the simple implementation takes $O(nm^k \log_2^k n)$ time. Comparing the total time against this, the break-even point occurs when n is of the order of m^2.

Suppose the coefficients are known ahead of time (perhaps computed once, stored in a database, and accessed in constant time.) Comparing the sum time as predicted by Theorem 3 against the simple implementation time the break-even point occurs when n is of the order of m.

5 Implementation

An implementation of the formulas is available in C, C++, Java, and Python. JDK does not provide the rational number APIs that are available in `gmp`. A minimal implementation has been rolled out as part of the Java implementation. The C implementation was written in somewhat object oriented fashion with the use of function pointers. There is little difference in execution time of C and C++ implementations. The code for the implementation is available in github. This code is thread-safe and one can build a library out of it and embed in a larger program. At present, the implementation was built and tested on Linux system.

5.1 Functions available

Table 7 lists the core functions available in all the implementations.

Table 7: Functions available in the implementation

Function name	Comment
`getCoefficients()`	To get the coefficients in the formula
`printSumFormula()`	To print the formula for $S_m(n)$ as a polynomial in n
`computeSum()`	To compute the sum for a given power and number of terms

The function arguments and return values are programming language specific.

5.2 Performance testing

To test the performance, a command line program was created which is available in the source distribution. A UNIX shell script (`runPerformanceTests.sh`) which is also available in the source distribution can run all the performance tests, capture the output in log files, process the log files, and use `gnuplot` to plot the performance data. We tested the performance on an Ubuntu system running on a Laptop with Intel® Core™ i5-8350U CPU @ 1.70 GHz. The

C implementation was used to run the performance tests. The plots of the perfomance test results are shown in the appendix.

5.3 Interpretation of results

Figure 1 shows the plot of coefficient initialization time for different m. The scale is logarithmic on both x and y axes. Asymptotically, except for the Central factorial formula, all the other ones have very close performance. The Bernoulli formula has slightly worse performance than others presumably because it requires more rational number operations. The Central factorial formula has the best performance. It has only $\lceil \frac{m}{2} \rceil$ terms and the computation involves only integer operations. Since the scales are logarithmic on both x and y axes, the plots are close to linear for all formulas as predicted by theorem Theorem 2

Figures 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and 13 show the plots for various m the total time and sum time versus n. Again the plots use logarithmic scaling for both axes. The total time includes time to initialize the coefficients and the time to compute the sum whereas the sum time is just the time to compute the sum. For comparison, the plot of the time to compute the sum with the simple implementation is also shown. It can be observed that the coefficient initialization time is about m times the sum time as predicted by Theorem 2 and Theorem 3. With logarithmic scaling, the sum time is pretty much flat and less sensitive to n. For Stirling, Euler, and Central factorial formulas, when $n < m$ only n coefficients are initialized and only n terms are computed in the formulas for the sum. Thus until $n = m$, the total time is much lower. It steadily increases and then flattens out after $n = m$. The break-even points are somewhat close to what are predicted by the theoretical analysis.

5.4 Code complexity

In order to calculate the code complexity, the tool Lizard [8] was used on the C code. C language was chosen since it uses minimal language specific libraries. The only library it uses is GNU `gmp`. The Faulhaber and Bernoulli formulas require multi-precision rational numbers in their implementations whereas the other formulas and the simple implementation just need multi-precision integers. The use of the `gmp` data type `mpq_t` (for multi-precision rational numbers) as opposed to just `mpz_t` (for multi-precision integers) may trigger extra logic inside the library. We are not accounting that in the complexity measure. Table 8 lists the number of lines of code (NLOC) and cyclomatic complexity number (CCN) for different implementations.

6 Conclusion

Overall, the CPU performance of Central factorial formula appears to be the best. In terms of code complexity, it is slightly higher due to the complexity of printing the sum formula. Historically well known formulas like Bernoulli and

Table 8: Code complexity

Formula	**NLOC**	**CCN**
Faulhaber	286	47
Bernoulli	175	30
Stirling	162	32
Euler	186	38
Central factorial	177	37
Simple implementation	14	4

Faulhaber have some disadvantages in terms of their implementations. They require rational number support. Faulhaber formula has the highest code complexity since there is no simple recurrence relation that defines the coefficients. In the end, we believe that the lesser-known Central factorial formula is the most suitable to program on a computer both in terms of code complexity and performance.

A Plots of performance test results

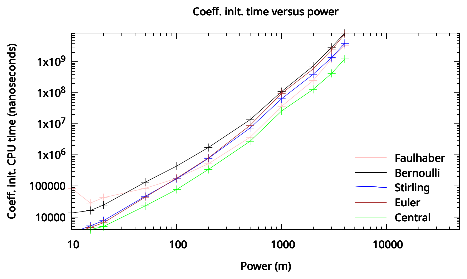

Figure 1: **Coefficient initialization time versus power**

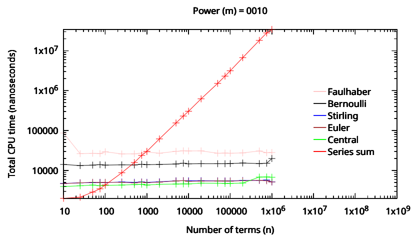

Figure 2: **Total time for** 10^{th} **power**

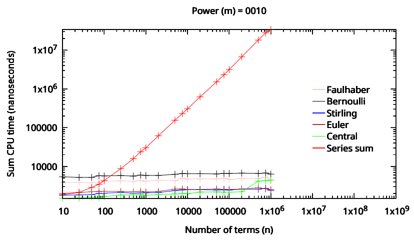

Figure 3: **Summation time for** 10^{th} **power**

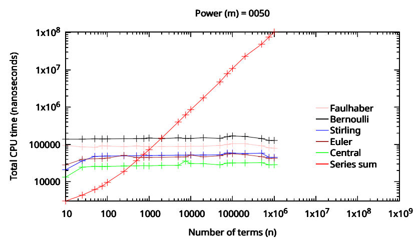

Figure 4: **Total time for** 50th **power**

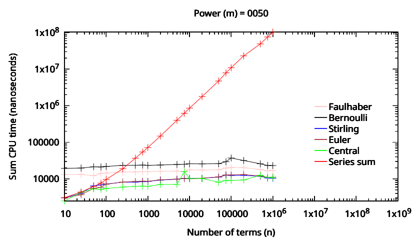

Figure 5: **Summation time for** 50th **power**

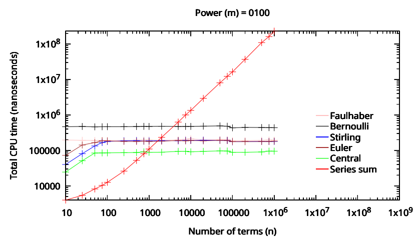

Figure 6: **Total time for** 100th **power**

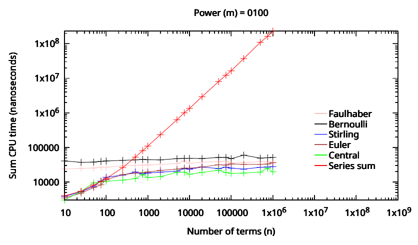

Figure 7: **Summation time for** 100th **power**

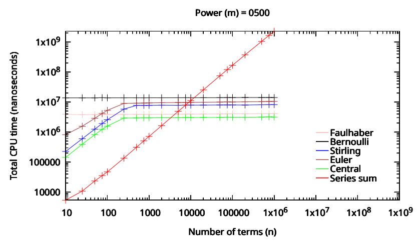

Figure 8: **Total time for** 500th **power**

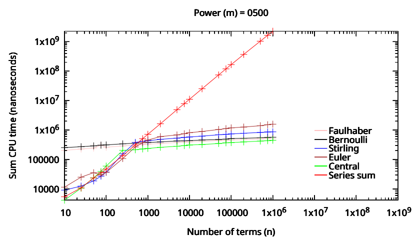

Figure 9: **Summation time for** 500th **power**

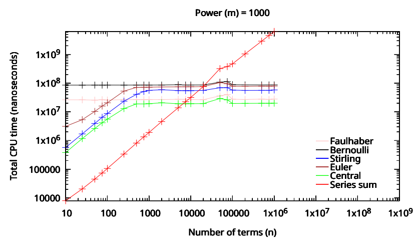

Figure 10: **Total time for** 1000th **power**

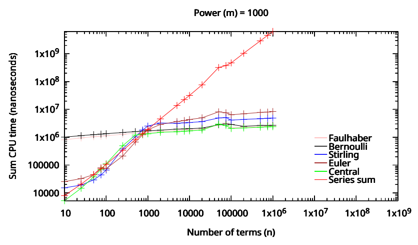

Figure 11: **Summation time for** 1000th **power**

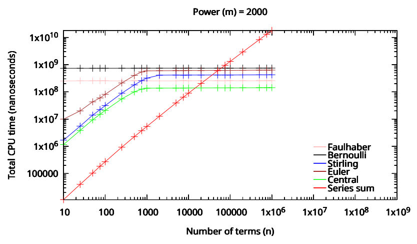

Figure 12: **Total time for 2000^{th} power**

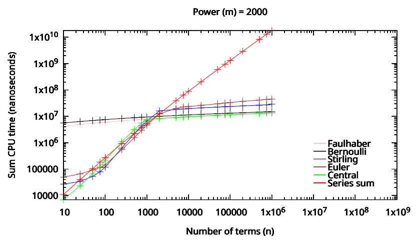

Figure 13: **Summation time for 2000^{th} power**

References

[1] Anthony William F Edwards. A quick route to sums of powers. *The American mathematical monthly*, 93(6):451–455, 1986.

[2] Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, second edition, 1994.

[3] Torbjörn Granlund and the GMP development team. GNU Multi-precision Library, 2020.

[4] Donald E. Knuth. Johann faulhaber and sums of powers. *Mathematics of Computation*, 61(203):277–294, 1993.

[5] John Riordan. *Combinatorial identities*. Wiley series in probability and mathematical statistics. Wiley, New York, 1968.

[6] Wikepedia. Bernoulli number, 2025.

[7] Wikepedia. Hockey-stick identity, 2025.

[8] Terry Yin. Lizard, 2025.