MOSI Algorithm: An Efficient Approach for Generalized Power Sum Computation in Arbitrary Arithmetic Sequences

Power Sum Computation Bernoulli Numbers Regression Line Arithmetic Sequence Lagrange Interpolation Fast Fourier Transform FFT

$$S_k(n) = \sum_{i=1}^{n} \left(a + (i-1) \cdot d\right)^k$$

Abstract

This paper introduces the MOSI algorithm, a novel and efficient method for calculating power sums over arbitrary arithmetic sequences, including those with non-integer and negative common differences. Unlike traditional approaches that rely on Bernoulli numbers and are limited to natural number sequences, MOSI generalizes the computation to accommodate any arithmetic sequence. By transforming the power sum problem into a regression problem, the algorithm significantly improves computational efficiency and broadens applicability. MOSI leverages regression analysis, Lagrange interpolation, and FFT to handle various arithmetic sequences effectively, making it suitable for applications in number theory, engineering, and data science.

Algorithm And Complexity

The MOSI algorithm operates in four key steps:

1. **Generate data points**: Compute the k-th power of the first $k+1$ terms of an arithmetic sequence, obtaining $k+1$ two-dimensional coordinates. For an arithmetic sequence with starting value a and step d, the generated points are

$$(a, a^k), (a+d, (a+d)^k) \dots (a+kd, (a+kd)^k)$$

2. **Linear regression**: Calculate the regression line slope and intercept for the first 2 points, first 3 points, and so on, up to the first $k+1$ points, yielding slope array b and intercept array a. The slope array b is a polynomial of degree $k-1$ with respect to x, and the intercept array a is a polynomial of degree k with respect to x. These are solved iteratively using the least squares method. The time complexity is $O(k)$.

$$b = \frac{\sum_{i=1}^{n} x_i \cdot y_i - n \cdot \bar{x} \cdot \bar{y}}{\sum_{i=1}^{n} x_i^2 - n \cdot \bar{x}^2}$$
$$a = \bar{y} - b \cdot \bar{x}$$

3. **Lagrange interpolation or FFT**: Obtain the functional forms of the slope array b and the intercept array a, denoted by $F_b(x)$ and $F_a(x)$, using Lagrange interpolation or the Fast Fourier Transform (FFT). The final regression line is obtained. The time complexity is $O(k^2)$ or $O(k \log k)$.

$$f(x) = F_b(n) \cdot x + F_a(n)$$

4. **Constructing the power sum formula**: Using the results from the interpolation, the algorithm reconstructs the general power sum formula for the given arithmetic sequence.

$$\begin{aligned} x_1^k + x_2^k + \ \dots \ + x_n^k &= f(x_1) + f(x_2) + \ \dots \ + f(x_n) \\ &= [F_b(n) \cdot x_1 + F_a(n)] \\ &+ [F_b(n) \cdot x_2 + F_a(n)] \\ &+ \ \dots \\ &+ [F_b(n) \cdot x_n + F_a(n)] \\ &= F_b(n) \cdot (x_1 + x_2 + \cdots + x_n) + F_a(n) \cdot N \end{aligned}$$

$x_1, x_2 \dots x_n$ is an arithmetic sequence with the first term x_1 and a common difference d, N is the number of terms in the arithmetic sequence.

$$\begin{aligned} N &= \frac{n - x_1}{d} + 1 \\ &= \frac{n}{d} + (1 - \frac{x_1}{d}) \\ x_1 + x_2 + \ldots + x_n &= (x_1 + n) \cdot \frac{N}{2} \\ &= \frac{n^2}{2 \cdot d} + \frac{n}{2} + (\frac{x_1}{2} - \frac{x_1^2}{2 \cdot d}) \end{aligned}$$

Expand it into the form of a general term formula, and thereby obtain the coefficients of each term.

$$\begin{aligned} F_a(n) &= a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_0 \\ F_b(n) &= b_{k-1} \cdot n^{k-1} + b_{k-2} \cdot n^{k-2} + \ldots + b_0 \\ x_1^k + x_2^k + \ \dots \ + x_n^k &= F_b(n) \cdot (x_1 + x_2 + x_3 + \ldots + x_n) + F_a(n) \cdot N \\ &= (b_{k-1} \cdot n^{k-1} + b_{k-2} \cdot n^{k-2} + \ldots + b_0) \cdot [\frac{n^2}{2 \cdot d} + \frac{n}{2} + (\frac{x_1}{2} - \frac{x_1^2}{2 \cdot d})] \\ &+ (a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_0) \cdot [\frac{n}{d} + (1 - \frac{x_1}{d})] \end{aligned}$$

Time Complexity

- **Step 1 (Generating points)**: This step involves computing the first $k + 1$ powers of the sequence terms, resulting in a time complexity of $O(k)$.
  Step 2 (Linear regression): For each subset of points, the regression line is fitted using the least squares method. The time complexity is $O(k)$, as the number of points and iterations increases linearly with k .
- **Step 3 (Lagrange interpolation or FFT)**: Using FFT to solve the interpolation problem reduces this step's complexity to $O(k \log k)$.

- **Step 4 (Constructing the power sum formula)**: In this step, each term in the summation is multiplied by the corresponding terms in $F_b(n)$, and the two terms in N are multiplied by each term in $F_a(n)$. Given that the number of operations scales linearly with k, the overall time complexity for this step is $O(k)$.
- **Overall**: The total time complexity of the algorithm is $O(k \log k)$, significantly improving over traditional methods for large values of k .

Space Complexity

- The algorithm requires $O(k)$ space to store the points, slopes, and intercepts, resulting in an overall space complexity of $O(k)$.

Example

To calculate the general term coefficients of $1^3 + 2^3 + 3^3 + \cdots + n^3$:

1. **Calculate the slope and intercept of the first four points:**
   For points $(1, 1^3)$ and $(2, 2^3)$, the regression line coefficients are: $a = -6$, $b = 7$
   For points $(1, 1^3)$, $(2, 2^3)$ and $(3, 3^3)$, the regression line coefficients are: $a = -14$, $b = 13$
   For points $(1, 1^3)$, $(2, 2^3)$, $(3, 3^3)$ and $(4, 4^3)$, the regression line coefficients are: $a = -27$, $b = 20.8$
   For points $(1, 1^3)$, $(2, 2^3)$, $(3, 3^3)$, $(4, 4^3)$ and $(5, 5^3)$, the regression line coefficients are: $a = -46.2$, $b = 30.4$
   The following values are obtained after performing the regression analysis.

x	2	3	4	5
a	-6	-14	-27	-46.2
b	7	13	20.8	30.4

2. **Derive the slope and intercept functions** using Lagrange interpolation or the Fast Fourier Transform (FFT):

$$F_b(n) = 0.9 \cdot n^2 + 1.5 \cdot n + 0.4$$
$$F_a(n) = -0.2 \cdot n^3 - 0.7 \cdot n^2 - 0.7 \cdot n - 0.2$$

3. **Substitute into the formula** to calculate the result. This example outlines the steps to compute the general term coefficients using regression and interpolation methods.

$$
\begin{aligned}
x_1^k + x_2^k + x_3^k + \ldots + x_n^k &= F_b(n) \cdot [\frac{n^2}{2 \cdot d} + \frac{n}{2} + (\frac{x_1}{2} - \frac{x_1^2}{2 \cdot d})] + F_a(n) \cdot [\frac{n}{d} + (1 - \frac{x_1}{d})] \\
&= (0.9 \cdot n^2 + 1.5 \cdot n + 0.4) \cdot [\frac{n^2}{2 \cdot 1} + \frac{n}{2} + (\frac{1}{2} - \frac{1^2}{2 \cdot 1})] \\
&+ (-0.2 \cdot n^3 - 0.7 \cdot n^2 - 0.7 \cdot n - 0.2) \cdot [\frac{n}{1} + (1 - \frac{1}{1})] \\
&= (0.9 \cdot n^2 + 1.5 \cdot n + 0.4) \cdot [\frac{n^2}{2} + \frac{n}{2}] \\
&+ (-0.2 \cdot n^3 - 0.7 \cdot n^2 - 0.7 \cdot n - 0.2) \cdot n \\
&= (0.9 \cdot \frac{1}{2} - 0.2) \cdot n^4 + (0.9 \cdot \frac{1}{2} + 1.5 \cdot \frac{1}{2} - 0.7) \cdot n^3 \\
&+ (1.5 \cdot \frac{1}{2} + 0.4 \cdot \frac{1}{2} - 0.7) \cdot n^2 + (0.4 \cdot \frac{1}{2} - 0.2) \cdot n \\
&= \frac{1}{4} \cdot n^4 + \frac{1}{2} \cdot n^3 + \frac{1}{4} \cdot n^2
\end{aligned}
$$

Conclusion

The MOSI algorithm provides an efficient and general solution for computing power sums over any arithmetic sequence, surpassing traditional approaches in terms of applicability and computational complexity. By leveraging regression analysis, Lagrange interpolation, and FFT, the algorithm is capable of handling sequences with non-integer and negative steps, making it suitable for a wide range of applications. Future work could explore further optimization techniques or extend the method to other forms of sequences.

Compare

Performance Comparison between Bernoulli Algorithm and MOSI Algorithm

Characteristic	Bernoulli Algorithm	MOSI Algorithm
Applicability	Mainly applicable to natural number arithmetic sequences	Applicable to any real-number arithmetic sequence (including non-integer and negative differences)
Time Complexity	$O(k^2)$, optimized to $O(k \log k)$	$O(k^2)$, optimized to $O(k \log k)$
Space Complexity	$O(k)$	$O(k)$
Recursiveness	Requires recursive calculation	Direct computation without recursion
Symbolic Computation	Primarily supports numerical computation	Supports both symbolic and numerical computation
Numerical Stability	May suffer from precision issues	Improved stability with symbolic computation

Use Cases	Limited to specific sequences and sum problems	Broadly applicable to any arithmetic sequence

Potential Applications of the MOSI Algorithm

Application Area	Potential Uses of MOSI Algorithm
Number Theory and Combinatorics	Analysis of power sums over arbitrary arithmetic sequences, particularly with symbolic computation
Engineering and Physics	Efficient computation of power sums with non-integer or negative differences, useful for modeling systems
Computer Science and Data Analysis	Applied to analyze large datasets where sequence sums are needed, particularly in generating closed-form solutions
Finance	Fast calculation of power sums in financial models, such as compound interest calculations, producing analytical results

Code

https://github.com/android-notes/1-x-2-x-...-n-x/blob/develop/sum.py

Case: 1^100+2^100+...+n^100 =
1/101*n^101 + 1/2*n^100 + 25/3*n^99 + -2695/2*n^97 + 298760*n^95 + -66698170*n^93 +
43232541100/3*n^91 + -2987368590010*n^89 + 592545208316600*n^87 +
-1909010939318560231/17*n^85 + 6092762457626069950/3*n^83 +
-10503770178403996919537/3*n^81 + 574696979476592789539800*n^79 +
-897017248688518684047572l0*n^77 + 1329674547053092686390429685 2*n^75 +
-18692982397686184162341538132 90*n^73 + 2488709557519908472602708844074 00*n^71 +
-10652456867712692797849086136518280 05/34*n^69 +
37236524072975827276192748905919310 75*n^67 +
-10844299000116828980379757772973769420469/26*n^65 +
43950288418050613210495571589828389262800*n^63 +
-43484475056945854288391661851382234152496 84*n^61 +
40313971117917025173667025724848011164192660 0*n^59 +
-34944260063316672143127900206016265956506516820*n^57 +
28253938453143723168879635164637743022201838744 80*n^55 +
-46975128963737486419489164499794297560673231041202090/221*n^53 +
14838677083702274079364344314958348779821717660137900*n^51 +
-95846360770205570066295244225555815970836313167084513 0*n^49 +
57102248319760803358389861613269821873306861123018474800*n^47 +
-31271532234285015125722229547830173546982502121910919455 72*n^45 +
15683919868422022059506295476842438409966285757387376139220 0*n^43 +
-93273006623793637434656479802977293641893710056414115793830132/13*n^41 +
29805522211776744798869487577678870257530893145282867254229640 0*n^39 +
-38042068156278908133943662769774849861948660969613013824505454 7645/34*n^37 +
37751106925714396719731413688661517086578640876419637362626864 9635*n^35 +

-22758671683254934243234770245768111655371809025564559292966948184145/2*n^33 +
3043834930058664295159051399208565084955170571695981686113075417146 00*n^31 +
-93215398532963113031284566771666289746833192047887884379325411297276490/13*n^29 +
14748253739612060527064121409232091921866498429836849523091519936602530 0*n^27 +
-13112574861745373977119865065563790341936971501269597493806710307275562234/5*n^25 +
39857448167724145521712375634202721882854746999642509687138048831663608600*n^23 +
-86845874263440736064895047098830501706784184056356720104299841572776660840 23/17*n^21
+ 16304634906294224258925539055711850026937478314146500355475130833028845746350/3*n^19
+
-61206781825168683912074666805758320400301005930260100721160098213379239450929 5/13*n^17
+ 32438843349198494485241907592045062093186546263622613006394068836873473016332 0*n^15
+
-17255395521678131518076029729610477618614494438827320896722047322110866375905 30*n^13
+
22501098457349089604835813048032888179174787302866854155337786876776064731954 0900/33*n^11
+
-56995948223784756021697384698478769187481110881793715780290371409596995575985270/3*n^9
+
34649381621107623485288357725441359210991177541965976048926944361209977534643400*n^7
+
-16293234618989521508515025064456465992824384487957638029599182473343901462949018943/442*n^5
+
18674771685049011296614057325260991542019103825338734064995339343748060441015725*n^3
+
-94598037819122125295227433069493721872702841533066936133385696204311395415197247711/33330*n
+ 0

Author

Email: 772478760@qq.com
Github: https://github.com/android-notes

Update Date 2024-10-15