1:
Hello everyone.
This video describes a method to enumerate magic squares efficiently.
2:
For a positive integer N, a magic square of order N is defined as an NxN square grid of distinct integers in the range from one to N^2, where the sum of the integers in each row,
3:
each column,
4:
and both diagonals is the same. This sum is called the ‘magic sum’, and its value is given by this formula $(1+2+\dots N^2)/N = (N + N^3)/2$.
5:
How many magic squares are there? That is a long-standing problem, and no simple formula or easy way to find the number has been discovered so far.
For more than fifty years, 5 was the highest order of magic square whose enumeration had been completed. In 2024, I completed the enumeration for order 6, and this video describes the essential ideas I used in the calculation.
6:
Let us start by introducing some representations of a square gird of distinct numbers.
At this stage of the discussion, we don’t care about the magic sum constraint, and any permutation of numbers are allowed. We call this square grid of distinct numbers a ‘distinct number square.’
A magic square is a distinct number square with additional constraints.
7:
The simplest representation of a distinct number square is a 2-dimensional array or a matrix, which is essentially a mapping from a pair of row-column indices to an element value. Mathematically speaking, there is no need to introduce another representation. Computationally, however, an alternative representation may offer advantages in certain contexts.
8:
We introduce a new representation, which we call the ‘Row sets-Column sets representation.’
A row set is the set of integers which belong to a row. A column set is defined similarly for a column. Since all elements in a distinct number square are unique, Row sets have no elements in common with each other - and the same holds for the column sets. The structure of a square imposes an important constraint: each row set have exactly one element in common with each column set.
9:
Given such row sets and column sets, we uniquely define a distinct number square. From a matrix representation we can straightforwardly derive the row sets column sets with no ambiguity. Conversely, from the Row sets Column sets representation, we can reconstruct all elements of the matrix representation by this formula, again without ambiguity. Thus, the two representations have a one-to-one correspondence and are equivalent.
10:
Next, we introduce the binary coding of a distinct integer set. In this scheme, each bit of a binary word represents the presence or absence of an element, as demonstrated in this example. This coding is a natural and convenient representation of a set of distinct objects. Set operations like intersection, union, or complement can be performed by bitwise logical operations.
11:
We call the Row sets-Column sets representation with the binary coding the ‘binary coded Row sets Column sets representation,’ or BRC representation for short.
12:

Adopting BRC representation is expected to offer certain advantages in the enumeration of magic squares.
First, it requires fewer primitive data objects to represent a magic square. BRC representation needs only 2N words, in contrast to N^2 in the matrix representation. Although BRC requires longer words, modern CPU's and GPU's typically support 64-bit word size, which enable us to use the BRC representation for magic squares up to order 8. BRC can leverage the wide bit-length of current CPU's and GPU's by introducing a degree of parallelism in word-level operations.
Second, row and column sum constraints are easily incorporated in the BRC representation. We can prepare a list of conformant row sets and column sets in advance, and, for the most part of the calculation, forget about the constraints.
In addition, constraints between rows and columns can be efficiently checked using bitwise logical operations.
Finally, BRC representation offers an advantage when working with the M-transformations.
13:
M-transformations are permutations of rows and columns that transform one magic square into another.
Here is an example for N=6. If we swap rows R2 and R5, and columns C2 and C5, the resulting square is still a magic square.
14:
Here is another such transformation - and there are more.
15:
The rule of M-transformation is to permute rows and columns conjointly, or to apply the same permutation to both rows and columns. Additionally, the permutation must be symmetric with respect to the centerlines which are depicted here.
The multiplicity of M-transformation is easily calculated, as shown here. These numbers do not include the multiplicity arising from simultaneous horizontal and vertical reflections, because the result is equivalent to a 180-degree rotation.
16:
Since an M-transformation consists of permutations of rows and columns, all magic squares generated by such transformations correspond to the same combination of row sets and column sets.
If we enumerate magic squares by generating combinations of row sets and column sets, we may find many magic squares generated by M-transformations in a set.
17:
We have discussed the advantages of the BRC representation in the enumeration of magic squares.
However, it is not immediately clear how to incorporate the diagonal sum constraints into this approach. This non-trivial question is at the heart of our discussion, and we will explore its solution step by step.
18:
Before tackling the diagonal constraints, let us consider what happens if we ignore them.
If we remove the diagonal constraints, semi-magic squares will be included in the enumeration.
19:
Semi-magic squares satisfy all the row and column constraints but not necessarily the diagonal ones.
As a matter of terminology, some people include magic squares as a subset of semi-magic squares, while others treat them separately.
In the following, terminology choice aside, we enumerate both semi-magic and magic squares inclusively.
20:
And it should be noted that the inclusive enumeration of semi-magic and magic squares is easier than the exclusive enumeration of magic squares alone.

21:
As mentioned earlier, we can prepare in advance a candidate list for row sets and column sets for constructing magic or semi-magic squares. The members of this list are called magic series.
A magic series of order N is a set of N distinct integers in the range from 1 to N^2 whose sum is equal to the magic sum. Here are some examples of magic series. Despite the name, a magic series is an unordered set of integers. The order of elements has no significance.
22:
Here is a table showing the number of magic series for several small values of N. For order 6, the candidate list has approximately 32,000 entries.
23:
Now we are ready to enumerate semi-magic and magic squares inclusively. If we pick up row sets and column sets from the list of magic series and generate all possibilities that conform the given constraints, we obtain all semi-magic and magic squares. Practically, however, we do not need to generate all possibilities. Permuting rows or permuting columns of a semi-magic square arbitrarily results in another semi-magic square.
24:
Therefore, it is sufficient to generate only distinct combinations of row sets and column sets under this constraint and then multiply the final count by $(N!)^2/4$. The division by 4 partially compensates the duplication due to rotations and reflections.
In addition, exchanging all row sets with all column sets produces another trivial duplication. To eliminate this, we add an extra constraint R1 > C1.
25:
Using an optimized code implementing this algorithm, I computed the number of combinations for N=6, and obtained this result.
By multiplying this count by the symmetry factor just described, I obtained this final result - which exactly matches the value obtained by A. Ripatti using a completely different method in 2018.
26:
The code, running on an NVIDIA RTX-4090, can complete the enumeration in approximately 10,000 hours, or about 14 months. If we use multiple GPUs on cloud services, the task can be completed within a few months.
Now the question is:
Can we incorporate the diagonal constraints into this approach without prohibitive overheads?
The answer is yes - as we will see later in this video.
27:
According to the inclusive definition of a semi-magic square, all magic squares are semi-magic.
28:
So, in principle, if we generate all semi-magic squares and check whether each one satisfies the diagonal constraints, we can obtain the correct number of magic squares. However, generating all semi-magic squares is not a feasible task. Although we succeeded in enumerating semi-magic squares, that was only possible thanks to the large multiplicity factor $(N!)^2/4$.
29:
What we actually generated were the distinct combinations of row sets and column sets, not the full set of semi-magic squares.
30:
To incorporate the diagonal constraints, We introduce two magic series X1 and X2 as diagonal candidates.
They must have only one element in common with each row and with each column. In addition, X1 and X2 must share no elements when N is even, and exactly one element when N is odd. To eliminate a part of trivial duplicate we impose the technical constraint X1 > X2.

31:
Given a combination of row sets R's, column sets C's, and diagonal candidates X's, we cannot yet be sure that any magic square exists. However, the row sets and column sets define a fixed semi-magic square, and the elements of X1 and X2 can be placed into the square unambiguously according their intersections with the rows and columns.
32:
This figure shows a hypothetical example of how X1 and X2 might be arranged. This is not a magic square. But there may exist permutations of row sets and column sets that place the elements of X1 and X2 along the diagonal lines, resulting in a valid magic square. However, as mentioned earlier, trying all permutation is computationally infeasible. We have to determine whether magic squares can be formed without testing many permutations. How can we do it?
33:
As a consequence of the intersection conditions, the elements of each diagonal candidates appear exactly once in each row and in each column. This means that an arrangement of a diagonal candidate corresponds to a permutation of N objects, or an element of the symmetric group S_N. In this example of order 6, observe the positions of X1. Read the row indices of X1, from left to right. We obtain: 4, 1, 3, 2, 6, 5. That sequence represents the permutation defined by the arrangement of X1. For convenience, we denote the permutation corresponding to X1 as d, and the one corresponding to X2 at this stage as u.
34:
Although trying all permutations is infeasible, a limited set of systematic permutations may be helpful. In particular, by permuting the columns in a specific way, we can always align x1 diagonally. This figure shows the result of such a column permutation to the example from the previous slide. This operation can be regarded as d^{-1} because it maps the permutation d to the identity. As a result of this operation, the arrangement of X2 is transformed from u to ud^{-1}.

Our next goal is to align X2 along the upward diagonal line. However, we no longer have full freedom to permute rows and columns independently. To keep X1 within the diagonal line, any further rearrangement must be a conjoint permutation – that is, the same permutation applied to both rows and columns. Can we still align X2 along the upward diagonal line under this constraint? In this example shown in the figure, the answer is no. Can you guess why? We will soon see a simple criterion that give us the answer.
35:
We are now discussing how the arrangement of a diagonal candidate, which corresponds to a permutation, is transformed by a conjoint permutation. Let X be the permutation corresponding to a diagonal candidate, and let P be a columns permutation. Then, the conjoint permutation transforms X into $P^{-1}XP$, which is known as a conjugation in group theory.
So our question becomes: Is the arrangement of X2 or ud^{-1} conjugate to the upward diagonal permutation?
36:
With this translation, we can now take advantage of a well-known theorem from group theory: Two permutations are conjugate if and only if they have the same cycle type.
37:
Then, let us examine the cycle type of the upward diagonal permutation, or reverse-order permutation. As an example, here is the upward diagonal permutation of order 6. In this permutation, 1 maps to 6, and 6 maps to 1 – so they form a 2-cycle. Likewise, 2 and 5 and 3 and 4 form 2-cycles. In general, the upward diagonal permutation consists of as many 2-cycles as possible and – for odd N – exactly one unpaired 1-cycle.
38:
Now, our task is to examine the cycle type of the permutation corresponding to X2, which is ud^{-1}.

Here is another piece of good news. Because of the intersection constraints we imposed between the two diagonals, no elements of X2 appear on the diagonal line when N is even. This implies that ud^{-1} has no 1-cycles in that case. For odd N, exactly one element of X2 appears in the downward diagonal line, resulting in one 1-cycles. So, the 1-cycle component is already guaranteed to be correct.

39:

With this fact, the criterion for identifying magic square becomes simple: ud^{-1} must have no cycles longer than 2. In other words, if ud^{-1} is squared it must be equal to the identity.

40:

We are now at the final stage of the discussion and are going to dive into a deeper and more concrete level of the calculation. To check whether this equation $(ud^{-1})^2 = e$ holds, we define permutation functions. For a permutation p, we define a function p with the same name such that p(i) returns the permuted result of i. With this function, the equation to be checked is equivalent to the set of these. Though it may seem necessary to check all N equations from 1 to N, we actually don't have to – and how far you need to go is an interesting question, which we will explore shortly.

41:

Next, we specify how to compute the function u and d^{-1}. u(i) is the row index of the element of X2 that lies in the i-th column. And $d^{-1}(i)$ is the column index of the element of X1 that lies in the i-th row. To calculate these values in BRC representation, we introduce functions Intersec, Elem2row, and elem2col.

42:

Intersec takes two binary-coded sets and returns the unique element in their intersection. Elem2row takes an element value and returns the row index where the element is located. Elem2col returns the column index. Using these functions, u and d^{-1} can be computed with these expressions.
Implementing the Intersec function is straightforward: Perform a bit-wise AND operation on the two binary coded sets, and identify the position of the bit set to 1 using CLZ or a similar instruction.
The elem2row and elem2col functions are not so obvious. A practical solution is to maintain small lookup tables that map element values to their corresponding row or column indices. See the sequel video for details of this implementation.

43:

The discussion so far applies to arbitrary order N. Now, let us see how we can tune the program for the concrete case of order 6. As we saw earlier, the criterion for magic squares has the following form: $u(d^{-1}(u(d^{-1}(i)))) = I$, i=1..N. A bit surprisingly, in the case of order 6, it is sufficient to check only two of these equations to determine the existence of magic squares.
First, we arbitrarily choose index 1, and check whether it returns to 1 after four-folds application of these permutations.
If it fails, we can immediately deny the existence of magic squares and return false. If it succeeds, we move to the next check with index 2. However, if the partner of index 1 happens to be index 2, checking index 2 gives no additional information. In that case, we take another index, say 3 instead.

If these two independent checks pass, then we have confirmed the existence of two of 2-cycles involving 4 distinct indices. This means only two indices remain – and they cannot form a 3-cycle or anything longer.
So we can safely conclude that the permutation composed only of 2-cycles, and thus magic squares exists.

44:

Here is a pseudocode of the check function, which lies at the deepest level of the program. It returns a boolean indicating the existence of magic squares for a given combination of rows, columns and diagonal candidates.

45:

Note that, there are no loops, no function calls, no large tables, and no side effects. It's an ideal solution enabling fast execution across a wide range of hardware platforms.

Now, we are done.
46:
But I would like to offer a few additional comments on the graphical patterns of diagonal candidate that lead to magic squares. Before diagonalizing either diagonal candidates, it is difficult to detect any clear pattern that predict which configuration will result in a magic square.
47:
However once we diagonalize X1, the arrangement of X2, or the permutation ud^{-1} begin to exhibit a noticeable order.
48:
This figure shows all patterns of X2 after X1 has been diagonalized that lead to magic squares. Remarkably, all of them are symmetric. And this symmetry can be explained as follows.
49:
Take this configuration as an example, and replace every element of X2 with the value 1, and all other elements with 0.
50:
This form a permutation matrix, which we will denote by P. As discussed earlier, for the configuration to yield a magic square, the permutation must satisfy: $P^2 = I$, where I is the identity matrix. This condition implies $P = P^{-1}$. And since P is a permutation matrix – hence orthogonal – it follows that $P = P^T$. In other words, P must be symmetric.
51:
I wrote a program that implements the ideas explained in this presentation. By leveraging the power of GPU, optimizing and refactoring the code, tuning it repeatedly I made the enumeration of 6x6 magic squares feasible.
52:
My code essentially generates the combinations of magic series for rows, columns, and diagonal candidates that satisfy the conditions we discussed, and counts the cases in which the check function returns true.
53:
The result for N=6 is this number.
54:
At this point, we must remember the role of M-transformation. All magic squares generated by an M-transformation share the same combination of row sets, column sets, and both diagonals. So, Every time the check function returns true, we are actually identifying a set of magic squares generated by the M-transformations.
55:
Therefore, we multiply the count by 24 to obtain the total number of 6x6 magic squares:
56:
The enumeration was performed twice to detect and correct hardware errors. In total, the full calculation took approximately 2 years of real time, using extensive GPU resources on cloud services.

As a performance measure of the code, running the code on a single RTX-4090, would take around 54,000 hours, or 6 years and 3 months. This is only about 5.4 times longer than the time required to enumerate semi-magic squares of order 6 – a remarkably small factor, considering the additional constraints.
57:

There are many important details and techniques – not explained here – that were essential to making the enumeration of 6x6 magic squares feasible.
58:
For those, please refer to the sequel videos. For references and technical notes, see the description text of this video.
59:
Thank you for watching. See you then.