卡儿的数学库 v.1.14

对应MC版本1.20.4

相关概念：万进制数组、分段存储、浮点型、double型、前导0、绝对值、常数、精度、科学记数法

如果万进制数组中的元素不足四位，则读数时应向前补0补足四位

本数据包里的世界实体、展示实体、临时实体等都在主世界

存档文件夹下data文件夹里的command_storage_large_number.dat文件便是本数据包产生的所有storage数据存储的位置。

推荐设置：`gamerule maxCommandChainLength 2147483647`

♦ 常数

```
圆周率 π：storage large_number:const "π"
自然常数 e：storage large_number:const "e"
欧拉常数 γ：storage large_number:const "γ"
黄金比例 ϕ：storage large_number:const "ϕ"
非数 NaN：storage large_number:math buffer_NaN
```

♦ 六个基本三角函数：large_number:math_trifs/_of_entity

```
输入：entity b09e-44-fded-6-efa5ffffef64 Rotation[0] 0.0f
输出：#sin int，#cos int，#tan int，#cot int，#sec int，#csc int
```

♦ 正弦与余弦

```
输入：entity b09e-44-fded-6-efa5ffffef64 Rotation[0] 0.0f

计算：execute as b09e-44-fded-6-efa5ffffef64 rotated as @s rotated ~ 0.0
positioned .0 .0 .0 run tp @s ^1.0 ^ ^ ~ ~

sin：entity b09e-44-fded-6-efa5ffffef64 Pos[2]

cos：entity b09e-44-fded-6-efa5ffffef64 Pos[0]
```

♦ 双参数反正切 (atan2d)：

公式：atan2d(y,x)

1.数据来自记分板：large_number:math_trifs/atan2

```
输入：#y int，#x int
计算：as b09e-44-fded-6-efa5ffffef64 run func..
输出（角度）：entity b09e-44-fded-6-efa5ffffef64 Rotation[0]
```

2.数据来自nbt：`execute as b09e-44-fded-6-efa5ffffef64 positioned .0 .0 .0 run function large_number:math_trifs/atan2_double/start with storage large_number:math atan2_double`

```
输入：
y：storage large_number:math atan2_double.y 1.0
x：storage large_number:math atan2_double.x 1.0
输入可以是double或float，输出的是float

输出（角度）：storage large_number:math atan2_double.output
```

♦ 反正弦与反余弦

反正弦：large_number:math_trifs/arcsin

反余弦：large_number:math_trifs/arccos

公式：`arcsin(x)=atan2(x,√(1-x²))`，`arccos(x)=atan2(√(1-x²),x)`

```
输入：#arcsin_cos.input int
放大一万倍输入，输入范围：[-10000,10000]

输出（角度）：entity b09e-44-fded-6-efa5ffffef64 Rotation[0]
```

♦ 反正切：`execute as b09e-44-fded-6-efa5ffffef64 positioned .0 .0 .0 run function large_number:math_trifs/arctan/start with storage large_number:math arctan`

公式：`arctan(x)=atan2(x,1)`

```
输入：storage large_number:math arctan.input 0.0
输入可以是double或float，输出的是float

输出（角度）：storage large_number:math arctan.output
```

♦ 大数加法：large_number:addition/start

```
加数1：storage large_number:math addition.input1 [I;0,0,0]
加数2：storage large_number:math addition.input2 [I;0,0,0]
和：storage large_number:math addition.output
```

♦ 大数减法：large_number:subtraction/start

```
被减数：storage large_number:math subtraction.input1 [I;0,0,0,0]
减数：storage large_number:math subtraction.input2 [I;0,0,0,0]
差：storage large_number:math subtraction.output
```

♦ 展示实体法浮点数除法

注：由于矩阵SVD，若输入值都是正数则输出的是必为正数，若输入值是负数则输出值不一定为正或负

```
输入    列表里第1、6、11个是被除数，最后一个数是除数
entity 28529-0-3d00-0-2c4200ee8401 transformation
[1.0f,0.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,0.0f,0.0f,1.0f
]

输出
entity 28529-0-3d00-0-2c4200ee8401 transformation.scale
```

♦ 展示实体法大数除法：large_number:division/display_large_number/start

仅处理正数

```
被除数
storage large_number:math display_div_large.input.dividend1 [I;0,0,0]
storage large_number:math display_div_large.input.dividend2 [I;0,0,0]
storage large_number:math display_div_large.input.dividend3 [I;0,0,0]

除数
storage large_number:math display_div_large.input.divisor [I;0,0,0]

输出
entity 28529-0-3d00-0-2c4200ee8401 transformation.scale
```

♦ 浮点除法 - 数据来自记分板

1. 八位有效数字：large_number:division/hpo/_div

这是目前所有高精度除法的核心，算法著作人：小豆 https://github.com/xiaodou8593

```
设置被除数
#float_sign int (符号，可选：-1，0，1，分别表示负号，0，正号)
#float_int0 int (输入值的前八位有效数字。取值为10000000~99999999或0)
#float_exp int (指数，范围是全int)

设置除数
#Divisor_float_sign int (符号，可选：-1，0，1，分别表示负号，0，正号)
#Divisor_float_int0 int (输入值的前八位有效数字。取值为10000000~99999999或0)
#Divisor_float_exp int (指数，范围是全int)

示例：
set #float_sign int 1
set #float_int0 int 44553375
set #float_exp int 23
则表示的数为：1*0.44553375*10^23

以改变被除数的方式输出
```

2. 12位有效数字：large_number:division/multi_times_modulo

```
被除数
#float_sign int (符号，可选：-1，0，1，分别表示负号，0，正号)
#float_int0 int (输入值的前八位有效数字。取值为10000000~99999999或0)
```

```
#float_exp int (指数，范围是全int)

除数
#Divisor_float_sign int (符号，可选：-1，0，1，分别表示负号，0，正号)
#Divisor_float_int0 int (输入值的前八位有效数字。取值为10000000~99999999或0)
#Divisor_float_exp int (指数，范围是全int)

商：
#float_sign int (符号)
#float_int0 int(前八位) + #float_int1 int(九~十二位)
#float_exp int (指数)

若#float_int1 int的分数不足四位数，则读数时应在数的前面补0补足四位
```

♦ 浮点除法 - 数据来自nbt

8位有效数字：large_number:division/float/start

12位有效数字：large_number:division/float_12dicimal/start

皆可输入float或double型

```
被除数：storage large_number:math float_division.input1 0.0
除数：storage large_number:math float_division.input2 0.0
商：storage large_number:math float_division.output
```

♦ 12位数组除以常数 (保留四位小数)：large_number:division/list_div_const

原理：竖式除法

输出的数组的第四个数是小数，常数不能超过五位数。

```
输入：
被除数：storage large_number:math list_div_const.dividend [I;0,0,0]
除数：#list_div_const_divisor int

输出：
商：storage large_number:math list_div_const.output
商的正负号：storage large_number:math list_div_const.output_sign
```

♦ 无穷多位有效数字的除法：large_number:division/loop_more_more_dicimal/start

```
被除数
#float_sign int (符号，可选：-1，0，1，分别表示负号，0，正号)
#float_int0 int (输入值的前八位有效数字。取值为10000000~99999999或0)
#float_exp int (指数，范围是全int)

除数
#Divisor_float_sign int (符号，可选：-1，0，1，分别表示负号，0，正号)
#Divisor_float_int0 int (输入值的前八位有效数字。取值为10000000~99999999或0)
#Divisor_float_exp int (指数，范围是全int)

有效数字的位数：#loop_more_more_dicimal_times int

商：
```

```
#float_sign int (符号)
storage large_number:math loop_more_more_dicimal_base (底数)
#float_exp int (指数)

输出的底数是个列表，读数方式是把每个元素从前往后写出来，在最前面加上0.
比如我得到的#float_sign int的值是1，#float_exp int的值是12，底数是[0,0,1,9,0,3,7,0]
则它们表示的数字就是1*0.00190370*10^12
```

♦ 对浮点数取倒数：large_number:division/float_reciprocal/start

可输入float或double型

```
输入：storage large_number:math float_reciprocal.input 0.0
输出：storage large_number:math float_reciprocal.output
```

♦ 整数除法

4位有效数字：large_number:division/int_4dicimal/start

8位有效数字：large_number:division/int_8dicimal/start

12位有效数字：large_number:division/int_12dicimal/start

作为浮点除法的推广，虽然可接受全int，但实际上只取被除数和除数的前八位

```
被除数：#int_+dicimal.input1 int
除数：#int_+dicimal.input2 int

商：storage large_number:math int_more_dicimal_out
```

♦ 数组除以整数 (多位有效数字)：large_number:division/list_div_int/start

被除数必须为万进制int数组，被除数的数组元素和除数必须全都是正数。有自适应数位，被除数数组不必每次都输入满三个数。

只取除数的前八位

原理：分段除法，(a+b+c)/m = a/m+b/m+c/m

无迭代，无试除，无递归，命令数固定

```
被除数：storage large_number:math list_div_int.list [I;0,0,0]
除数：storage large_number:math list_div_int.int 1

商 (double型)：storage large_number:math list_div_int.output
```

♦ 对整数进行任意倍乘：large_number:int_mul_by_n/start

原理：execute store + data get，可实现用倍率存储整数，用函数宏导入动态倍率

```
输入整数：storage large_number:math int_mul_by_n.input_int
输入倍率：storage large_number:math int_mul_by_n.input_n
要输出的数据类型：storage large_number:math int_mul_by_n.data_type "double"

输出：storage large_number:math int_mul_by_n.output
```

输入的"整数"可以为非整数，但会按照整数来处理，向下取整并把范围钳制在整型范围内
输入的"倍率"可以为任何数值，但计算时会忽略数据单位并转化为double型
可选的数据类型："byte"、"float"、"double"、"short"、"int"、"long"

♦ 浮点乘法

算法1：large_number:float_multiply/start

原理：execute store + data get，可实现用倍率存储整数，用函数宏导入动态倍率

算法2：large_number:float_multiply/of_score/start

原理：浮点转化为记分板格式后取前八位进行大数乘法

```
因数1：storage large_number:math float_multiply.input1 0.0
因数2：storage large_number:math float_multiply.input2 0.0
可以为float或double型

积：storage large_number:math float_multiply.output
```

♦ 高精度浮点乘法：large_number:float_mul.high_precision/start

原理：采用了全新架构，用double转int数组的算法把输入值全都转化成数组然后进行大数乘法，再根据读出来的输入值的信息计算指数

可精确到浮点数级

```
因数1：storage large_number:math float_multiply.input1 0.0
因数2：storage large_number:math float_multiply.input2 0.0
可以为float或double型

积：storage large_number:math float_multiply.output
```

♦ 高精度浮点数平方：large_number:float_mul.high_precision/squ/start

```
输入：storage large_number:math float_multiply.input1 0.0
可以为float或double型

输出：storage large_number:math float_multiply.output
```

♦ 浮点加减法：large_number:float_add_subtra/start

输入可以是float或double型，但是输出的一定是double型

原理：execute positioned + loot spawn，用函数宏输入参数。loot spawn无坐标上下限，故此算法可以计算全浮点数的加减法。

```
输入：
storage large_number:math float_add_subtra.input1 0.0
storage large_number:math float_add_subtra.input2 0.0

计算模式：set #float_add_subtra_ope_mode int
1为加法，2为减法
若是加法，则为两数相加，若为减法，则是input1减input2

输出：storage large_number:math float_add_subtra.output
```

♦ 浮点数比大小：large_number:float_comparison_sizes/start

把输入值代入浮点减法，判断输出值的符号

```
输入：
storage large_number:math float_comparison_sizes.A 0.0
storage large_number:math float_comparison_sizes.B 0.0

输出比较结果：storage large_number:math float_comparison_sizes.output
"A"比较"B"，"+"为更大，"-"为更小，"="为相等
```

♦ 对浮点数取整：large_number:round_double/start

execute align+实体tp只能处理区间 (-30000000.0, 30000000.0) 的数，而此算法采用了函数宏+字符串递归找小数点的方法，可以处理全部浮点数

```
输入：storage large_number:math round_double.input 1.0
可以是float或double

向0取整：set #round_towards_zero int 1
此值不为1就是向下取整，默认是向下取整

输出：storage large_number:math round_double.output
```

♦ 对浮点数进行10进制位移：large_number:double_displacement/decimal.start

```
输入：storage large_number:math double_displacement.input 1.0
可以是double或float

位移的次数：storage large_number:math double_displacement.shift 2
可以是任意整数

输出：storage large_number:math double_displacement.output
```

♦ 任意整型数字相乘：large_number:int_int_multiply

原理：数组乘法，竖式相乘

```
因数1：input int
因数2：input.2 int
积：storage large_number:math int_int_multiply.output
```

♦ 任意整型数字平方：large_number:int_square

```
输入：input int
输出：storage large_number:math int_squ
```

♦ 12位数字相乘：large_number:1we_multiply

```
因数1：storage large_number:math 1we_multiply.input1 [I;0,0,0]
因数2：storage large_number:math 1we_multiply.input2 [I;0,0,0]
积：storage large_number:math 1we_multiply.output
```

♦ 12位数字平方：large_number:1we_square

```
输入：storage large_number:math 1we_squ.input [I;0,0,0]
输出：storage large_number:math 1we_squ.output
```

♦ 无穷位数字相乘：large_number:infinite_digit_multiply/start

```
因数1：storage large_number:math Infinite_digit_multiply.input1 [I;0,0]
因数2：storage large_number:math Infinite_digit_multiply.input2 [I;0,0]
输入格式：因数必须为万进制int数组，且数组元素全都是正数

输出：storage large_number:math Infinite_digit_multiply.output
```

♦ 整型数字拆分为数组：large_number:cut_math_to_list

```
输入：input int
输出：#sign int (符号)，#1st int，#2nd int，#3rd int
```

♦ 整型数字开方：

取整 (16条纯记分板命令)：large_number:int_sqrt_simple

保留四位小数 (32条纯记分板命令)：large_number:int_sqrt

保留多位小数：large_number:test_int_more_dicimal

开1~5位，保留9位；开6~7位，保留8位；开8~10位，保留7位

有时求得的最后一位小数会有稍许的精度损失

如果保留小数位数不足期望的位数，则读数时应在数的前面补0补足数位

原理：初值预估+牛顿迭代，详见参考文献

```
输入：input.sqrt int

取整输出：output.sqrt int
保留四位小数输出(放大一万倍)：output.sqrt int

保留多位小数的输出：
整数部分：output.sqrt int
小数部分：output.dicimal int
```

♦ 整型数字开方 - 连分数迭代法：large_number:sqrt_continued_fraction/start

精确度可达14位小数。
连分数迭代法的小数部分是以分数形式输出的。
内置溢出检查，可在分子/分母其中一个溢出前自动停下。
例如在计算√10时，迭代50次和11次的输出是一样的。
因分子分母都是以单段计分板存储，所以实际可允许的迭代次数不超过32次。
使用前建议了解一下什么是连分数。

连分数开根号公式：

$$\sqrt{x} = \frac{x-\lfloor\sqrt{x}\rfloor^2}{2\lfloor\sqrt{x}\rfloor+\frac{x-\lfloor\sqrt{x}\rfloor^2}{2\lfloor\sqrt{x}\rfloor+\frac{x-\lfloor\sqrt{x}\rfloor^2}{2\lfloor\sqrt{x}\rfloor+\ldots}}}$$

此为无限连分数，算的层数越多越接近。

```
被开方数：#conti_frac.sqrt.input int
迭代次数：#conti_frac.sqrt.loops int
约分：set #conti_frac.sqrt.reduction_fraction int 1
显示连分数表达式：set #conti_frac.sqrt.tellraw int 1

输出：
整数部分：#conti_frac.sqrt.inte int

小数部分：
分子：#conti_frac.sqrt.A int
分母：#conti_frac.sqrt.N int

连分数表达式：storage large_number:math conti_frac_sqrt_expression
```

♦ 整型数字开方 - 牛顿迭代法 (保留四位小数)：large_number:newton.s_method_sqrt/int_dicimal.4

以数组除以常数为思路，无试除，无递归，无二分树，41条纯记分板命令

```
输入：#Newton's-method_sqrt.input int
输出(放大一万倍)：#Newton's-method_sqrt.output int
```

♦ 10~16位数字开方

原理：高精度猜测法。只对前八位数开方算结果的前四位。结果的后面几位用估小数的算法来算

取整：large_number:large_sqrt_digit16

估值法取小数：large_number:large_sqrt_digit16_with_dicimal

竖式法取小数：large_number:large_sqrt_digit16_vertical_method

```
输入：storage large_number:math large_sqrt_digit16.input [I;0,0,0,0]

高精度模式：set #large_sqrt16.test16 int 1

输出：
整数部分：storage large_number:math large_sqrt_digit16.output
小数部分：storage large_number:math large_sqrt_digit16.output_dicimal
整数和小数两部分合并：storage large_number:math
large_sqrt_digit16.output_with_dicimal
```

高精度模式是16位整数开方算法的特性，为了追求高效率选用了高精度猜测法，代价是最后一位会有稍许的精度损失。仅在处理16位数的时候会有这种特性。
高精度模式就是通过平方根自我平方对比原数来验证大小，自己决定要不要开。估值法取小数默认开启高精度模式。

而竖式法取小数是采取无精度波动的竖式开方法，但只能取出四位小数

♦ 1~24位数字开方 (取整)：large_number:large_sqrt

原理：牛顿迭代+竖式开方

```
输入：storage large_number:math large_sqrt.input [I;0,0,0,0,0,0]
输出：storage large_number:math large_sqrt.output
```

为了避免浪费算力，请按照如下优先级使用：整型范围内选整型数字开方，10~16位数字选16位数字开方，最后再考虑24位数字开方。

♦ 整型数字求立方根

原理：立方根估值算法。取一个常数x，n是x的立方根整数部分，z是立方根小数部分，则 (x-n^3)/(3n^2+3n+1)≈z。整数部分是二分法。

取整：large_number:cube_root/floor

保留四位小数：large_number:cube_root/4dicimal

```
输入：#cbrt.input int
输出：#cbrt.output int
若保留四位小数则放大一万倍输出
```

♦ double的欧氏范数

输入的数据类型必须为double型，只接受正值

1.二维范数

三角函数法：`execute as b09e-44-fded-6-efa5ffffef64 run function large_number:double_norm/trif_2d`

公式：`√(x²+y²)=x/cos(atan2(y,x))`

单位向量法：`execute as b09e-44-fded-6-efa5ffffef64 run function large_number:double_norm/unit_vector_2d`

```
输入：
storage large_number:math double_norm_2d.x 1.0d
storage large_number:math double_norm_2d.y 1.0d

输出：storage large_number:math double_norm_2d.output
```

2.三维范数

三角函数法：`execute as b09e-44-fded-6-efa5ffffef64 run function large_number:double_norm/trif_3d`

公式：`√(x²+y²+z²)=λ/cos(atan2(z,λ))`，其中λ是关于x和y的二维范数

单位向量法：`execute as b09e-44-fded-6-efa5ffffef64 run function large_number:double_norm/unit_vector_3d`

```
输入：
storage large_number:math double_norm_3d.x 1.0d
storage large_number:math double_norm_3d.y 1.0d
storage large_number:math double_norm_3d.z 1.0d

输出：storage large_number:math double_norm_3d.output
```

♦ double转int - 数组格式，精度为16位有效数字：large_number:double_to_int

对float型数值也有效

```
输入：storage large_number:math double_to_int.input 0.0d
输出：storage large_number:math double_to_int.output
```

参数介绍：`math` 是尾数，16位int万进制数组。`sign` 是符号，byte型，取整为1或-1。`expon` 是指数，short型。

读数方法：以S*0.AEB形式读数，S是符号，A是尾数，B是指数。

示例：`{sign:1b, math:[I;1623,13,3007,6000], expon:2s}` 表示的数为 `1*0.1623001330076000*10^2`，也就是`16.23001330076`。

附：SNBT的浮点数规律

对于每一个数字，必定存在符号和数值。对于MC里的浮点数，指数、小数点位置和前导0数量这三个信息并不会同时变动，若其中一个变了，其他两个参数一定是固定值。也就是说，对于转化后的数字信息：
如果指数不为0，则小数点位置必定为2(在第一个数后面)，前导0必定是0个。
SNBT的浮点数整数部分达到8位或小数的前导0数量多于3个就会以科学记数法形式显示。
如果小数点位置不为2，则指数必定为0，前导0必定是0个。
如果前导0数量为1到3个(MC浮点数最多存在三个前导0)，则小数点位置必定为2，指数必定为0。

此外，SNBT的浮点数也可以以科学记数法的形式输入，比如1.2E3d，以科学记数法形式输入时必须带数据单位。

♦ double转int - 记分板格式，精度为8位有效数字：large_number:float_nbt_to_score

```
输入：storage large_number:math float_nbt_to_score_input 0.0

输出：
符号：#float_sign int
尾数：#float_int0 int
指数：#float_exp int

示例：
#float_sign int 1
#float_int0 int 44553375
#float_exp int 23
则表示的数为: 1*0.44553375*10^23

转换后的尾数始终是八位
```

♦ double型开方 (高精度浮点数开方)

对float型数值也有效

8~9位有效数字：large_number:double_sqrt

12~14位有效数字：large_number:double_sqrt_more_dicimal

用24位数组开根法取出了double开根号的12位有效数字

"8~9位有效数字"的命令数约为180，"12~14位有效数字"的命令数约为1430，后者的消耗约为前者的8倍。

```
输入：storage large_number:math double_sqrt.input 0.0d
输出：storage large_number:math double_sqrt.output
```

♦ 快速浮点数开方：large_number:new_double_ope/double_sqrt

新架构牺牲了一点精度，采用了性能更佳的算法
基础59条命令，如果输入的是科学记数法则加12条，如果选择精度增加四位则加9条，最多80条命令

原理：使用放大倍率存储法来获取double的底数，使用字符串取数法来获取指数。用整型开方法算结果后根据指数来调整输出。

```
输入：storage large_number:math double_sqrt.input
可输入double型/float型
精度增加四位：set #New_double_sqrt.dicimal_add int 1

输出：storage large_number:math double_sqrt.output
```

♦ 24位数字显示

输入几位就显示几位：large_number:digital_display
始终保持显示的数字是24位：large_number:24_digital_display
区别：后者如果输入的数字不足24位，则会自动在数字前面补0补足24位

每三位数一组用逗号隔开。若数组中任意一个数为负数，则视为整个数组为负

```
输入(万进制数组)：storage large_number:math math_display [I;0,0,0,0,0,0]

显示以下JSON文本便可显示数字：
[{"nbt":"math_display_json_is-","storage":"large_number:math"},
{"nbt":"math_display_json[]","storage":"large_number:math","separator":
{"text":","}}]
```

♦ 单位向量法测距

1.输入任意两点：large_number:unit_vector_for_distance
两个点的坐标差的范围：100*|x|+100*|y|+100*|z| ≤2147483

```
输入
P1： storage large_number:math unit_vector2.P1 [0.0,0.0,0.0]
P2：storage large_number:math unit_vector2.P2 [0.0,0.0,0.0]
运行：as b09e-44-fded-6-efa5ffffef64 run func...
输出(已放大10倍)：#distance int
```

2.输入两点坐标差的绝对值：large_number:unit_vector_for_distance_modu
需要玩家自己作差输入
输入值范围：100x+100y+100z ≤2147483

```
输入：storage large_number:math unit_vector_modu.input [0.0,0.0,0.0]
执行：as b09e-44-fded-6-efa5ffffef64 run func...
输出 (已放大10倍)：#distance int
```

♦ 三角函数法快速测距：large_number:fast_distance_trigonometry/start

算法来源：https://github.com/SuperSwordTW/Distance-Trig-Calc-3d

```
输入：#dx int，#dy int，#dz int
dy和dz值必须为正数

输出 (放大1000倍)：#distance int
```

♦ 列表算法 - 洗牌：large_number:list_operation/shuffle/start

随机打乱列表顺序

原理：@e[sort=random]

```
输入：storage large_number:math list_ope_shuffle.input []
输出：storage large_number:math list_ope_shuffle.output

清理列表算法产生的临时marker：
kill @e[type=minecraft:marker,tag=large_number.list_operation]
```

♦ 列表算法 - 抽牌：large_number:list_operation/random_index_once/start

从列表中随机抽取一个元素

原理：set from list[$(random)]

```
输入：storage large_number:math list_ope_random_index_once.input []
把抽到的项从原列表移除：set #list_ope_random_index_once.del int 1

输出：storage large_number:math list_ope_random_index_once.output
```

♦ 列表算法 - 元素去重 (返回值法)：large_number:list_operation/deduplicate/start

```
输入：storage large_number:math list_dedup.input []
输出：storage large_number:math list_dedup.output
```

♦ UUID数组转为带连字符的16进制：large_number:uuid_list_for_hyphen/start

例如：[I; 30583058, 20172024, 31415926, -3059] 转为 "01d2a912-0133-ccf8-01df-5e76fffff40d"

```
输入：storage large_number:math uuid_list_for_hyphen.input [I;0,0,0,0]
输出：storage large_number:math uuid_list_for_hyphen.output
```

♦ 带连字符的16进制UUID转为数组

算术法：large_number:uuid_list_for_hyphen/back

实体属性法：`function large_number:uuid_list_for_hyphen/back_for_attribute with storage large_number:math uuid_hyphen_back_list`

例如："00000035-ffff-f910-0000-00ffffffffd" 转为：[I; 53, -1776, 255, -3]
必须输入完整的32位UUID，每一段前面的0不能省
16进制UUID一共有32位，每一段的字符数固定为 8,4,4,4,12

```
输入：storage large_number:math uuid_hyphen_back_list.input ""
输出：storage large_number:math uuid_hyphen_back_list.output
```

♦ 概率模拟 - 二项分布

测试1：large_number:random/binomial_distribution/test1

测试内容：若输入值里包含2的幂，则有50%概率减去2的幂，从2^{30}到2^{0}测试31次，返回测试后的输入值

```
输入(只接受正值)：set #binomial_distribution.test1.input int
输出：#binomial_distribution.test1.output int
```

测试2：large_number:random/binomial_distribution/test2

测试内容：做n次成功概率为p的伯努利试验，测试一个[0, 10^9]之间的随机数是否小于给定值，输出成功次数

只接受正值，返回成功次数

试验次数不宜过多

```
试验次数：set #binomial_distribution.test2.n int
输入范围是[0,536870911]

给定值：set #binomial_distribution.test2.p int
单次试验的成功概率是p/(10^9)

输出：#binomial_distribution.test2.output int
```

当n足够大时，结果接近于正态分布。当n越大（至少20）且p不接近0或1时近似效果更好。不同的经验法则可以用来决定n是否足够大,以及p是否距离0或1足够远,其中一个常用的规则是np和n(1 −p)都必须大于 5。

♦ 概率模拟 - 正态分布：large_number:random/normal_distribution/test1/start

测试内容：输入上限值n，先生成一个int32的随机数，然后不断判断正负并x2，如果x2次数达到32次就再生成一个随机数继续这个操作，直到判断次数达到n次。然后把判断正负的结果(0或1)加起来，结果就趋近于0到n的正态分布。

```
上限值：set #normal_distribution.input int
输出：#normal_distribution.output int
```

♦ 概率模拟 - 均匀分布

此模块取自xwjcool写的NTRE数据包。

采用的是PCG算法，比Java自带的LCG算法漂亮一些。

随机范围是 -2147483648..2147483647

选定一个用于生成随机数的实体A：

```
初始化：as 实体A run func ntre:randomize
注：每个实体只需要在载入数据包时初始化一次

生成随机数：as 实体A run func ntre:next
结果输出在实体A的ntre_output记分板
```

♦ 概率模拟 - 超几何分布：large_number:random/hypergeometric_distribution/start

测试内容：从有限N个物件（其中包含M个指定种类的物件）中抽出n个物件，成功抽出该指定种类的物件的次数（不放回）。

```
样本池：storage large_number:math hypergeometric_distribution_list [1,2]
必须输入int型正整数列表。输入列表里的元素按照它所在的位置，自动分配ID。比如第1个元素的ID为1，第五个元素ID为5。每一项的数字表示这个ID的元素有几个。
要抽取的元素ID：#hypergeometric_distribution.target int
抽取次数：#hypergeometric_distribution.times int

输出：#hypergeometric_distribution.output int

清理测试产生的临时marker：
kill @e[type=minecraft:marker,tag=large_number.list_operation]
```

♦ 生成总和为n的a个随机数：large_number:random/sum_to_x/start

这里的总和求法是用的记分板的自带向上/向下溢出的加法

```
n：#random.sum_to_x.n int
a：#random.sum_to_x.a int
输出：storage large_number:math random_sum_to_x_out
```

♦ 指数函数

1. e^x： large_number:exp_e.x/start

e是自然对数的底，是一个无理数，e ≈2.718281828459045

例：输入 4.231123，输出 68.79444497242804

输入范围为区间：(-709, 709.7828)

```
需要载入前置库：function large_number:exp_e.x/database
卸载前置库：data remove storage large_number:exp database

输入：storage large_number:math exp_e^x.input 2.0d
输入值必须为double型

输出：storage large_number:math exp_e^x.output
```

2. 任意正数的幂：large_number:exp_any/start

原理：把指数拆为整数部分和小数部分，整数部分用快速幂，小数部分套公式，$a^b = e^{(b*\ln(a))}$ 。

例：输入 $5.7322^{2.1123}$，输出 39.97625953186048

指数范围：[0, 2147483647]

```
e^x的前置库：function large_number:exp_e.x/database

输入：
底数：storage large_number:math exp_any.input.base 2.0d
指数：storage large_number:math exp_any.input.expon 3.0d
输入值必须为double型

输出：storage large_number:math exp_any.output
```

♦ 整数的整数次幂：large_number:int_base_int_power/start

可计算负底数或负指数

指数范围：[-2147483647, 2147483647]

传统的递归相乘法

```
输入：
底数：#int_base_int_power.base int
指数：#int_base_int_power.expon int

输出：storage large_number:math int_base_int_power_out
```

♦ 浮点数的整数次幂：large_number:float_base_int_power/start

可计算负底数或负指数

指数范围：[-2147483647, 2147483647]

传统的递归相乘法

```
输入：
底数：storage large_number:math float_base_int_power.base 0.0
指数：storage large_number:math float_base_int_power.expon 0

输出：storage large_number:math float_base_int_power.output
```

♦ 浮点数的整数次幂 - 快速幂：large_number:float_base_int_power/fast_power/start

快速幂算法性能稳定，无论多大的指数，都最多使用30次浮点乘法和30次浮点平方，全面优于递归相乘法。

快速幂算法原理：https://baike.baidu.com/item/%E5%BF%AB%E9%80%9F%E5%B9%82

```
输入：
底数：storage large_number:math float_base_int_power.base 0.0
指数：storage large_number:math float_base_int_power.expon 0

输出：storage large_number:math float_base_int_power.output
```

♦ 整数的自然对数 ln(x)：large_number:ln/start

精度：误差不超过0.0009，保留四位小数

```
计算前需要载入初始数据库：function large_number:ln/ln_database

输入：#ln(x) int
输出(放大一万倍)：#ln(x).output int
double型输出：storage large_number:math ln_output

卸载初始数据库：function large_number:ln/uninstall_ln_database
```

♦ 对浮点数取自然对数 ln(x)：large_number:ln_double/start

对数公式：`ln(7.25)=ln(725/100)=ln(725)-ln(100)`，`ln(7.45*10^26)=ln(7.45)+26*ln(10)`

保留四位小数

```
计算前需要载入初始数据库：function large_number:ln/ln_database

输入：storage large_number:math ln_double.input 0.0d
输出(放大一万倍)：#ln_double.output int
double型输出：storage large_number:math ln_double.output
```

♦ 任意正整数的对数：large_number:loga.b/start

保留四位小数

换底公式：`log.a(b)=ln(b)/ln(a)`

特殊情况：
以0或1为底的"不为1的数"的对数不存在，故而输出的值也不存在；
任何数为底的1的对数都是0；
非0且非1的底数的0的对数都是负无穷，故而输出的double为负无穷，输出的计分板值是-2147483648。

```
计算前需要载入初始数据库：function large_number:ln/ln_database

输入：
底数：#loga.b_a int
真数：#loga.b_b int

输出(放大一万倍)：#loga.b.output int
double型输出：storage large_number:math "log.a(b).output"
```

♦ 对浮点数取对数：large_number:loga.b_double/start

```
计算前需要载入初始数据库：function large_number:ln/ln_database

输入：
底数：storage large_number:math log(a,b)_double.a 0.0
真数：storage large_number:math log(a,b)_double.b 0.0

输出：storage large_number:math log(a,b)_double.output
```

♦ 整数的常用对数：large_number:lg/start

保留四位小数

公式：`lg(x) = ln(x)/ln(10)`

```
计算前需要载入初始数据库：function large_number:ln/ln_database

输入：#lg(x) int

输出(放大一万倍)：#lg(x)_output int
double型输出：storage large_number:math lg(x)_output
```

♦ 高精度自然对数 (全double)：large_number:ln_high_precision/start

此算法参考：https://www.zhihu.com/question/333371020/answer/1686069171

雷米兹算法得到的多项式在高精度ln算法里起了最重要的误差修正的作用，理论上误差可低至2^-58.45。

此算法使用了大量的高精度浮点乘法，因此此算法的消耗约为查表法的60倍。

```
输入：storage large_number:math ln_high_precision.input 1.0
输入值必须为double型

输出：storage large_number:math ln_high_precision.output
```

♦ 自然数的阶乘：large_number:gamma_function/fundamental_factorial/start

输入范围为区间：[0,170]

区间[0,12]的自然数的阶乘以int型输出，区间[13,170]的自然数的阶乘以double型输出。

```
输入：#natural_num.factorial.input int
输出：storage large_number:math natural_num_factorial
```

♦ 自然数的双阶乘：large_number:gamma_function/fundamental_factorial/double_factorial

输入范围为区间：[0, 300]

区间[0,19]的自然数的双阶乘以int型输出，区间[20,300]的自然数的双阶乘以double型输出。

这里的双阶乘是原始的无穷乘积形式定义的

```
输入：#natural_num.double_factorial.inp int
输出：storage large_number:math natural_num_double_factorial
```

♦ 伽玛函数 - 斯特林公式：large_number:gamma_function/stirling/start

斯特林公式：

$$\Gamma\left(x+1\right) \sim \sqrt{2\pi x}\left(\frac{x}{\mathrm{e}}\right)^{x}\left(1+\frac{0.0845072303119}{x}\right)$$

Γ(x+1)在(-1, 0.2216) 区间的近似：

$$\Gamma\left(x+1\right) \sim \frac{1}{x+1}+\frac{25}{49}x$$

输入范围为区间：(-1, 170.6271]

这里计算的是Γ(x+1)，主要用于计算实数的阶乘

```
e^x的前置库：function large_number:exp_e.x/database

输入：storage large_number:math gamma_function.input 0.0d
输入值必须为double型

输出：storage large_number:math gamma_function.output
```

♦ 伽玛函数 - 递推公式：large_number:gamma_function/recursion/start

递推公式：

$$\Gamma\left(x+1\right)=x\Gamma\left(x\right)=\Gamma\left(x+1-a\right)\cdot\prod_{n=1}^{a}x+1-n,\ a\in\mathbf{N}$$

注：Π为连乘符号。a的取值取决于要把x钳制到哪个区间。

输入范围为区间: [0.001, 170.6026)

```
载入前置库：function large_number:gamma_function/recursion/database
卸载前置库：data remove storage large_number:math gamma_databse

输入：storage large_number:math gamma_function.input 0.0d
输入值必须为double型

输出：storage large_number:math gamma_function.output
```

♦ LambertW函数

LambertW(x)：large_number:lambertw/start
LambertW.(-1) (x)：large_number:lambertw/-1/start

LambertW(x)是x*e^x的反函数

公式1：`LambertW(x) ~ ln(x)-ln(ln(x))+ln(ln(x))/ln(x) x≥3`
公式2：`LambertW(x) ~ ln(x+1)/1.3 0≤x≤3`
公式3：`LambertW(x) ~ tan(3.365x)/3.2 (-1/e)≤x≤0`
公式4：`LambertW.(-1) (x) ~ ln(-x)-ln(-ln(-x))+ln(-ln(-x))/ln(-x)`

输入范围：
LambertW(x)：[-1/e, ∞)
LambertW.(-1) (x)：[-1/e, 0]
-1/e≈-0.3678794411714

要求输入值必须为double型

```
计算前需要载入初始数据库：function large_number:ln/ln_database

输入：storage large_number:math lambertw.input 1.0d
输出：storage large_number:math lambertw.output
```

♦ 逆伽玛函数 - F.K.Amenyou公式：large_number:inverse_gamma_function/start

这里计算的是Γ(x+1)的反函数，就是已知x的阶乘求x。

伽玛函数的函数值与x并不是单射关系，因此需要限制定义域。
取Γ(x+1)在x≥0的部分，可以发现这一段函数存在一个极小值λ，λ≈0.8856031944109。
定义一个常数φ，满足Γ(φ+1)=λ，φ≈0.4616321449684。
在[φ,∞)区间内，Γ(x+1)严格单调，所以在$x\in$[φ,∞)时，Γ(x+1)存在反函数。
定义隐式x=Γ(y+1) (y≥λ)，满足此关系式的点集就是正实数的反阶乘函数。称为逆Γ(x+1)，定义域为[λ,∞)。

F.K.Amenyou公式：

$$\text{逆}\Gamma\left(x+1\right)\sim\frac{\ln\left(\frac{x}{\sqrt{2\pi}}\right)}{\mathrm{LambertW}\left(\frac{\ln\left(\frac{x}{\sqrt{2\pi}}\right)}{\mathrm{e}}\right)}-\frac{1}{2}+\frac{1}{30x}$$

相关论文：https://ir.lib.uwo.ca/etd/5365/，https://www.ams.org/journals/proc/2012-140-04/S0002-9939-2011-11023-2/

逆Γ(x+1)在(λ, 1.13)区间的近似：

$$\text{逆}\Gamma\left(x+1\right)\sim\arcsin\left(1.23099326x-2.08932555\right)+\frac{\pi}{2}+\varphi$$

φ≈0.4616321449684，λ≈0.8856031944109

输入范围：x≥λ

```
输入：storage large_number:math inverse_gamma_function.input 1.0d
输出：storage large_number:math inverse_gamma_function.output
```

♦ 执行朝向转为四元数四分量xyzw：large_number:quaternion/facing/2tostoxyzw

需要传入执行朝向

```
执行：as b09e-44-fded-6-efa5ffffef64 run func...

输出：
列表形式：storage large_number:math xyzw
记分板分数：#qrot_x int, #qrot_y int, #qrot_z int, #qrot_w int
```

♦ 欧拉角转四元数：`execute as b09e-44-fded-6-efa5ffffef64 run function large_number:quaternion/euler_angles_to_xyzw`

```
输入：storage large_number:math euler_angles_input [0.0,0.0,0.0]
第一个是横滚(roll)，第二个是俯仰(pitch)，第三个是偏航(yaw)

输出：storage large_number:math xyzw
```

♦ 执行朝向转单位向量：large_number:quaternion/facing/facing_to_unit_vector

需要传入执行朝向

```
执行：as b09e-44-fded-6-efa5ffffef64 run func...
输出：storage large_number:math unit_vector
```

♦ 横滚角转四元数：`execute as b09e-44-fded-6-efa5ffffef64 run function large_number:quaternion/euler_angles_roll`

```
输入：storage large_number:math euler_angles_roll 0.0
输出：storage large_number:math xyzw
```

♦ 局部坐标转相对坐标

方法1 (向量点乘)：large_number:uvw/uvwtoxyz

需要传入执行朝向，需要以世界实体为执行者

```
输入：#u int，#v int，#w int
输出(放大一万倍)：#x int，#y int，#z int
```

方法2 (宏)：large_number:uvw/uvwtoxyz_2

输入执行坐标，执行高度(anchored eyes|feet)，执行朝向

需要以世界实体为执行者

```
输入：#u int，#v int，#w int
输出：#vec_x int，#vec_y int，#vec_z int
```

♦ 相对坐标转局部坐标

方法1 (向量点乘)：large_number:uvw/xyztouvw

需要传入执行朝向，需要以世界实体为执行者

```
输入：#x int，#y int，#z int
输出(放大一万倍)：#u int，#v int，#w int
```

方法2 (宏)：large_number:uvw/xyztouvw_2

输入执行坐标，执行高度(anchored eyes|feet)，执行朝向

需要以世界实体为执行者

```
输入：#vec_x int，#vec_y int，#vec_z int
输出：#u int，#v int，#w int
```

♦ 解整系数一元二次方程：large_number:quadratic_equation/start

需要把一元二次方程化为一般形式输入，a b c 的绝对值尽量不大于20724
支持a=0的情况

更精确的：支持的Δ的值的范围为全int，即 `-2147483648 ≤ b²-4ac ≤ 2147483647`

公式法求解：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
输入：
#X_squ_equ.a int
#X_squ_equ.b int
#X_squ_equ.c int

输出：
放大一万倍的分数形式：
#X_squ_equ.x1 int
#X_squ_equ.x2 int

表达式形式(未化简)：storage large_number:math quadratic_equation_out.expression
double型形式：storage large_number:math quadratic_equation_out.double

实数根的数量：#X_squ_equ.roots int

显示解方程的结果：set #X_squ_equ.tellraw int 1
显示这个JSON文本便可显示结果：
[{"nbt":"quadratic_equation_out_json_tellraw.json1","storage":"large_number:math
","interpret":true},
{"nbt":"quadratic_equation_out_json_tellraw.json2","storage":"large_number:math"
,"interpret":true}]
```

注：
1.若方程有两个不相等的实数根，则x1和x2的记分板分数都存在，表达式形式和double型形式都是列表，列表的第一项对应x1，第二项对应x2。
2.若方程有两个相等的实数根，则x1和x2的记分板分数都存在且相等，表达式形式是一个单独的字符串，double型形式是一个单独的double型数值。
3.若方程没有实数根，则x1和x2的记分板分数都不存在，表达式形式和double型形式也都不存在，`storage large_number:math quadratic_equation_out`会是一个空的复合标签。

♦ 获取当前日期和时间：large_number:timestamp/start

【此功能需要联网使用】

原理：解码正版玩家头颅里的Base64后会获得一个json对象，里面包含一个unix时间戳。

因获取玩家头颅里的Base64需要等待方块更新，所以解码会稍有延迟

已知bug：如果执行后，观察到执行后无输出，则表示头颅皮肤未正确加载，解决方法是延迟几tick再执行一次本函数
用命令判断就是测试此命令是否能通过，通过就表示解析不正确：`execute unless data storage large_number:timestamp output_base64_json.timestamp`

```
使用前需要载入前置库：function large_number:timestamp/database
卸载前置库：function large_number:timestamp/uninstall_database

输出
年：#timestamp_year int
月：#timestamp_month int
```

```
日：#timestamp_day int
时：#timestamp_Hour int
分：#timestamp_Minute int
秒：#timestamp_Second int

数位始终为两位的时分秒
时：storage large_number:timestamp output_day_Hour
分：storage large_number:timestamp output_day_Minute
秒：storage large_number:timestamp output_day_Second

显示以下JSON文本便可显示时间：
{"nbt":"output_base64_json_tellraw","storage":"large_number:math","interpret":true}

更换正版玩家ID：storage large_number:math player_head_cache_list ["<玩家名>","<玩家名>"]
注：
列表里可存多个玩家名，但读取时只读取列表里的第一个
初始输入的正版玩家ID：ka__er
```

因为每个正版玩家名仅能在进入单人存档/服务器时获取两次时间戳，一次是放置成方块，一次是放置在实体的物品栏里，然后时间戳就存在了缓存里不再更新，想要更新时间戳只有三个方法：
1.重进存档/重开服务器；2.一个月后头颅缓存自动过期；3.更换一个新的正版玩家ID
所以想要长期开着服务器，建议配合内部打表计时使用，每两小时用命令方块同步一次时间，每24小时更换一个新的正版玩家id来同步一次日期，更换30次后，第一次使用的玩家id的头颅缓存就过期了。

♦ Unix时间戳解析 (32位)：large_number:timestamp/parse_timestamp/start

```
输入 (可为整型或字符串)：storage large_number:math parse_timestamp.input

输入GMT时区：set #GMT-time_zone int 8
例如北京时间是GMT+8，所以输入8，默认为8

输出：
年：#parse_timestamp.year int
月：#parse_timestamp.month int
日：#parse_timestamp.day int
时：#parse_timestamp.Hour int
分：#parse_timestamp.Minute int
秒：#parse_timestamp.Second int

显示以下JSON文本便可显示解析结果：
{"nbt":"parse_timestamp.tellraw","storage":"large_number:math","interpret":true}
```

♦ 玩家经验公式 - 根据经验等级和经验数推出经验总数：
large_number:xp_formula/levels_to_points/start

当经验等级≥32时，玩家的经验数为：

$$f(x) = 1507 + \sum_{n=32}^{x-1} 9n - 158 \ = \ 4.5x^2 - 162.5x + 2099$$

输出的数值一般情况下不可直接用于逆推玩家已有的经验等级，因为mc内部的一些特殊算法，这个数与玩家此时真正拥有的经验数有些出入。
能差多少呢？举个例子："用xp命令一次性给予1628点经验"和"用xp命令分别给予一次1507点经验和一次121点经验"，玩家得到的经验数会差出1点。

原因是mc在计算玩家升级到下一级所需的经验数时使用了玩家nbt里的XpP参数，这是一个浮点型存储的百分比数，浮点误差导致了玩家实际拥有的经验与理论拥有的经验数略有出入。

```
输入：
等级：#xp_formula.levels int
经验数：#xp_formula.points int
经验数就是 /xp query @s points 获得的

输出：storage large_number:math xp.output
```

♦ 玩家经验公式 - 经验总数逆推经验等级和经验余数：large_number:xp_formula/points_ope_levels/start

当经验数大于等于1758时，逆推经验等级公式：

$$g(x) = \frac{\sqrt{72x - 45503} + 325}{18}$$

经验公式是个一元二次方程，对其用求根公式反推，然后只保留x≥0的根，得到了这个反向经验公式

理论上输入值不应大于 2.07526*10^19

```
输入：storage large_number:math xp_points_ope_levels.input [I;0,0,0,0,0]
本算法自适应位数，不必每次都输入满5个数

输出：
经验等级数：storage large_number:math xp_points_ope_levels.output_levels
经验余数：storage large_number:math xp_points_ope_levels.remaining_points

若用于给予玩家经验，应先给予经验等级再给予经验余数
```

♦ 颜色RGB转16进制：large_number:rgb_to_hexadecimal/start

```
输入(RGB值范围均为 0~255)：
#rgb_to_hexadecimal.R int
#rgb_to_hexadecimal.G int
#rgb_to_hexadecimal.B int

输出：storage large_number:math rgb_to_hexadecimal_output
```

♦ 调和级数前N项和：large_number:harmonic_series/sum1-n

公式法逼近，无递归。
公式：

$$H_x=\sum_{n=1}^{x}\frac{1}{n}=\psi(x+1)+\gamma\approx\ln(x)+0.5772+\frac{0.4995078}{x}$$

注：Σ为级数求和，ψ为Digamma函数，即伽玛函数的自然对数的导数，γ是欧拉-马歇若尼常数，也是调和级数的拉马努金和，约为0.5772156649

在输入值为负数时，输出5772，即调和级数的拉马努金和

```
计算前需要载入初始数据库：function large_number:ln/ln_database

输入：storage large_number:math Harmonic_series_sum_input 3.0
输入值的类型可以是：double/float/int，使用double/float型输入可以计算超出int范围的值

输出(放大一万倍)：#Harmonic_series.sum.output int
```

♦ Sigmoid函数 - 线性近似：large_number:sigmoid/start

Sigmoid(x)=1/(1+e^(-x))

原理参见：https://zhuanlan.zhihu.com/p/318423774

```
输入：storage large_number:math sigmoid.input 1.0
输出：storage large_number:math sigmoid.output
```

♦ Digamma函数：large_number:digamma_function/start

公式：`ψ(x)~ln(x)-1/(2x)`

在输入值为1时输出特殊值：-γ

```
ln的初始数据库：function large_number:ln/ln_database

输入：storage large_number:math digamma_function.input 0.0
输入值必须为double型，输入范围：x>0

输出：storage large_number:math digamma_function.output
```

♦ 整数质因数分解：large_number:prime_factorization/start

```
输入：#prime_factorization.input int
输出：storage large_number:math prime_factorization_output

如果输出的列表只有一项那么输入值就是一个质数
```

♦ 整数约分：large_number:int_simplify/start

原理：欧几里得算法，辗转相除法

只接受正数

```
输入值1：#int_simplify.input1 int
输入值2：#int_simplify.input2 int

约分后的输入值1：#int_simplify.output1 int
约分后的输入值2：#int_simplify.output2 int

两数的最大公约数：#int_simplify.greatest_common_divisor int
如果最大公约数为1，则两数互质
```

♦ 整数转二进制：large_number:convert_decimal_to_binary

条命令完成，无递归

按照32位有符号整数的存储规则进行转换，输出的列表为固定32个整数，每个整数表示这一位的二进制数，对于负数会进行补码

```
输入：#convert_decimal_to_binary.input int
输出：storage large_number:math convert_decimal_to_binary_out

显示以下JSON文本可显示输出结果：
{"nbt":"convert_decimal_to_binary_out[]","storage":"large_number:math","separato
r":""}
```

♦ 整数的进制转换

1. 10进制转2~36进制：large_number:number_base_conversion/10_to_any

```
输入：#conversion.10_to_any.input int
只接受正数

进制基数：#conversion.10_to_any.radix int
接受的进制基数为2~36

输出：storage large_number:math number_base_conversion
输出的是一个列表，列表的每一项表示在该进制下这一位的数
```

2. 2~36进制转10进制：large_number:number_base_conversion/any_to_10

```
输入：storage large_number:math number_base_conversion ["f","f","0","9","7"]

进制基数：#conversion.10_to_any.radix int
接受的进制基数为2~36

输出：#conversion.any_to_10.output int
```

♦ 表达式求值 - 四则运算

符号仅接受 `+－*/().E-` 。为了在转化为逆波兰式的过程中区分减法与负数，`－` (全角减号) 表示减法，`-` (半角减号) 表示负数。数字只能是int或double。double型数值可以是科学记数法且不需要单位，double型数值只能使用浮点数算法计算。

注：不要单独把一个数放在括号里，如有需求，请写成 (a+0) 的形式。此算法的表达式里没有 "负数要单独放在括号里" 这种规则。

逆波兰式算法：https://blog.csdn.net/zm_miner/article/details/115324206

转换完成与计算完成均有提示

1.表达式转换为逆波兰式：large_number:expression_evaluation/to_rev_polish_notation

```
输入：storage large_number:math expression_evaluation.input "(12+14)*(106－32)"
输出逆波兰式 (可直接用于解析求值)：storage large_number:math
expression_evaluation.rev_polish_notation
```

2.解析逆波兰式

使用整数算法来求值：large_number:expression_evaluation/ope_of_inte

使用浮点数算法来求值：large_number:expression_evaluation/ope_of_float

```
输入逆波兰式：storage large_number:math expression_evaluation.rev_polish_notation
["51E-2","3","+"]
输出计算结果：storage large_number:math expression_evaluation.output

显示逆波兰式 (JSON文本)：
{"nbt":"expression_evaluation.rev_polish_notation[]","storage":"large_number:mat
h","separator":" "}
```

♦ 表达式求值 - 科学计算

运算符可接受 `+－*·/().E-^²` (加减乘除、括号、小数点、科学记数法、负号、幂运算，平方)。`·` 等价于 `*` 。

对于幂运算，整数幂是递归相乘，非整数幂是查表算法。对于除法，若被除数为1，则执行专门的取倒数算法。

为了在转化为逆波兰式的过程中区分减法与负数，`－` (全角减号)表示减法，`-` (半角减号)表示负数。

数字只能是double，不需要带单位。

注：不要单独把一个数放在括号里，如有需求，请写成 (a+0) 的形式。此算法的表达式里没有 "负数要单独放在括号里" 这种规则。即使是变量与数字相乘，乘号也必须要写。

转换完成与计算完成均有提示。

函数列表 (已支持29种函数)：

> 每个函数和它的参数都必须单独放在一个括号里，支持复合函数。
> α、β和δ都是函数的参数，若参数为一个数字，则不应放在括号里，若参数不为一个数字，则应放在括号里。
> 例如sin7+2应写成"(sin7)+2"，ln(2+9)·2-3应写成"(ln(2+9))·2 - 3"

```
函数名称：exp; sin; cos; arcsin; arccos; arctan; ln; √; Γ; ᒪ; ⁰LambertW;
¹LambertW; ||; sgn; []; −; ψ; Σ[1/n]n→; log; atan; eunorm₂; eunorm₃; [0]; >=; <=;
==; ≈≈; >/<; >-<

介绍：

一元运算
expβ = e^β，指数运算，整数幂是递归相乘，非整数幂是查表算法。
sinβ = sin(β) 弧度制
cosβ = cos(β) 弧度制
arcsinβ = arcsin(β) 弧度制
arccosβ = arccos(β) 弧度制
arctanβ = arctan(β) 弧度制
lnβ = ln(β)，自然对数
√β = √β，平方根
Γβ = 伽玛函数，gamma(β)，输入范围为区间：(0, 171.6271]，对于整数是阶乘算法，非整数是斯特林公式。
ᒪβ = 逆伽玛函数，gamma(x)主分支的反函数，逆gamma(β)-1相当于阶乘的逆运算，输入范围：β≥λ，λ≈0.8856031944109。
⁰LambertWβ = LambertW⁰(β)，主分支，输入范围：[-1/e,∞)
¹LambertWβ = LambertW¹(β)，-1的分支，输入范围：[-1/e,0)
||β = β的绝对值
sgnβ = sgn(β)，符号函数
[]β = 把β向下取整
[0]β = 把β向0取整
−β = 破折号的一半，表示β的相反数。注：此符号与负号并不等价，此符号表示的是"取相反数"的函数。
ψβ = ψ(β) digamma函数，又叫双伽玛函数，伽玛函数的对数的导数
Σ[1/n]n→β = 调和级数前β项和，

二元运算
αlogβ = 以α为底β的对数
αatanβ = atan2(α,β) 弧度制
αeunorm₂β = √(α²+β²)，二维向量(α,β)的欧氏范数，必须都是非负数，计算方法是三角函数法。
α>=β = 逻辑运算，取较大值
α<=β = 逻辑运算，取较小值
α==β = 逻辑运算，严格判断是否相等，相等为1，否则为0
α>/<β = 交换除，β除以α
α>-<β = 交换减，β减α

三元运算
αeunorm₃β,δ = √(α²+β²+δ²)，三维向量(α,β,δ)的欧氏范数，必须都是非负数。此处的逗号仅作为把数字分开的占位符。计算方法是单位向量法。
α≈≈β,δ，逻辑运算，误差判断，判断α和β的距离是否在δ的绝对值以内，是为1，否则为0
注：可能会因浮点误差导致判断失误，例如0.02在计算时变为0.020000000000000018
```

需要的前置库：

```
e^x的前置库：
载入：function large_number:exp_e.x/database
卸载：data remove storage large_number:exp database

ln的初始数据库：
载入：function large_number:ln/ln_database
卸载：function large_number:ln/uninstall_ln_database
```

1.表达式转换为逆波兰式：large_number:expression_evaluation_scientific/to_rev_polish_notation

```
输入：storage large_number:math expression_evaluation.input "(12+14)*(106－32)"

支持代入变量，解析时可自动把变量视为指定路径的数字。对只有变量存在的式子也可解析，例如计算["π"]会输出3.141592653589793

目前支持的变量名：α; β; δ; ϵ; η; λ; μ; ξ; τ; ω; ｘ; ｙ; ｚ
此处的ｘｙｚ是全角字母
分别对应路径 (目标值只能是浮点数值):
α：storage large_number:math expression_evaluation_variables."α"
β：storage large_number:math expression_evaluation_variables."β"
δ：storage large_number:math expression_evaluation_variables."δ"
ϵ：storage large_number:math expression_evaluation_variables."ϵ"
η：storage large_number:math expression_evaluation_variables."η"
λ：storage large_number:math expression_evaluation_variables."λ"
μ：storage large_number:math expression_evaluation_variables."μ"
ξ：storage large_number:math expression_evaluation_variables."ξ"
τ：storage large_number:math expression_evaluation_variables."τ"
ω：storage large_number:math expression_evaluation_variables."ω"
ｘ：storage large_number:math expression_evaluation_variables."ｘ"
ｙ：storage large_number:math expression_evaluation_variables."ｙ"
ｚ：storage large_number:math expression_evaluation_variables."ｚ"

支持输入数学常数符号，解析时自动替换为对应数值：π，ｅ，γ
为了区分，此处应输入全角字母ｅ

输出逆波兰式 (可直接用于解析求值)：storage large_number:math expression_evaluation.rev_polish_notation
```

2.解析逆波兰式：large_number:expression_evaluation_scientific/ope

列表具有取出特定编号的项的功能，因此借助列表可以用逆波兰式定义非二元运算。

```
输入逆波兰式：storage large_number:math expression_evaluation.rev_polish_notation ["51ᴇ-2","3","+","°cos"]
输出计算结果：storage large_number:math expression_evaluation.output

显示逆波兰式 (JSON文本)：
{"nbt":"expression_evaluation.rev_polish_notation[]","storage":"large_number:math","separator":" "}
```

♦ 定积分

"表达式求值 - 科学计算" 的拓展

采用黎曼积分法，在区间里平均距离取样，把采样得到的值乘上小区间宽度。

只能求一重积分，被积函数在积分区间内必须"黎曼可积"，求出来的结果只能是个数（是数值积分，而且无法处理含参结果）。

被积函数直接取 "表达式求值 - 科学计算" 解析出来的逆波兰式，取积分变量为ｘ。

公式 (梯形法则):

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{k}\left(\frac{f(a)+f(b)}{2}+\sum_{n=1}^{k-1} f\left(a+\frac{b-a}{k}n\right)\right)$$

其中k是区间内小矩形的数量。这里的小矩形的高度取的是小区间右端的函数值。

[0,1]区间的积分：large_number:definite_integral/riemann_integral/0_1/start

其他区间的积分：large_number:definite_integral/riemann_integral/start

求解完成会有提示。

```
积分区域 下限(double)：storage large_number:math
expression_evaluation.definite_integral.a 1.0
积分区域 上限(double)：storage large_number:math
expression_evaluation.definite_integral.b 2.0
积分区间内小矩形的数量(int)：storage large_number:math
expression_evaluation.definite_integral.dx_times 200
取正整数，上限是1000000000，不宜太多，一般取100~500。

输出：storage large_number:math expression_evaluation.definite_integral.output
如果算完后此路径不存在，则表明计算量过大，超出了单tick的命令执行量，需要异步计算。
```

♦ 曲线长度 - 一元函数在[a,b]内的图像长度：large_number:curve_length/univariate_function/start

"表达式求值 - 科学计算" 的拓展

采用折线拟合的方法，只能处理连续函数

函数表达式直接取 "表达式求值 - 科学计算" 解析出来的逆波兰式

```
区域 下限(double)：storage large_number:math
expression_evaluation.definite_integral.a 2.0
区域 上限(double)：storage large_number:math
expression_evaluation.definite_integral.b 3.0
区间内取样数量(int)：storage large_number:math
expression_evaluation.definite_integral.dx_times 200
取正整数，上限是1000000000，不宜太多，一般取100~500。

输出：storage large_number:math expression_evaluation.univariate_function_length
```

♦ 数值导数

采用差商求导法

一阶导数中点公式：

$$f'(x_i) \approx \frac{f(x_i+\Delta x)-f(x_i-\Delta x)}{2\Delta x}$$

二阶导数公式：

$$f''(x_i) \approx \frac{f(x_i+\Delta x)+f(x_i-\Delta x)-2f(x_i)}{(\Delta x)^2}$$

所求导的函数直接取"表达式求值 - 科学计算"解析出来的逆波兰式

一阶导数值：large_number:differential/difference_quotient_method/1/start

二阶导数值：large_number:differential/difference_quotient_method/1/start

```
求导点的x值：storage large_number:math expression_evaluation.differential.input 1.0

Δx的大小：storage large_number:math expression_evaluation.differential.dx 0.04
Δx是一个较小的值，取值范围是[1, 1E-9]，因浮点误差的存在，此值不可太小，一般选0.01~0.001

一阶导数值：storage large_number:math expression_evaluation.differential.1output
二阶导数值：storage large_number:math expression_evaluation.differential.2output
```

♦ 三维空间任意方向的粒子圆

```
圆的半径(1000倍输入)：#3d.circle.r int
例如输入3000就是半径3

粒子密度：#3d.circle.angle int
粒子密度就是每隔"n/10"度描一个点，范围为1~3600

计算坐标：
execute as b09e-44-fded-6-efa5ffffef64 run function
large_number:particle/3d_ar_rotation_circle/start

输出相对坐标列表：
x：storage large_number:math 3d_ar_rotation_circle_posX
y：storage large_number:math 3d_ar_rotation_circle_posY

显示粒子：
execute positioned x y z rotated x y run function
large_number:particle/3d_ar_rotation_circle/particle
执行朝向就是圆的朝向，执行位置就是圆的原点
```

```
把圆染色成色环：function large_number:particle/rainbow_circle/start

输出颜色列表：storage large_number:math rainbow_circle_color

显示染色后的圆：
execute positioned x y z rotated x y run function
large_number:particle/rainbow_circle/particle.macro1

通过旋转颜色列表可以实现霓虹灯那样的轮转闪烁效果，这是一个例子：
初始化：
data modify storage large_number:math rainbow_circle_color_list_rotate set from
storage large_number:math rainbow_circle_color
显示粒子：
execute positioned x y z rotated x y run function
large_number:particle/rainbow_circle/particle_list_rotate
```

♦ 三维空间任意方向的五角星

两个算法均出自：https://www.bilibili.com/read/readlist/rl651851

算法一：公式法绘制

```
半径(100倍输入)：#3d.pentagram.r int
例如输入500就是半径5

粒子密度：#3d.pentagram.density int
粒子密度就是每隔"n/10"度描一个点，范围为1~3600

五角星的横滚角(1000倍输入)：#3d.pentagram.roll.θ int

计算坐标：
execute as b09e-44-fded-6-efa5ffffef64 run function
large_number:particle/3d_ar_rotation_pentagram/start
```

算法二：摆线法绘制

就是把高频的盔甲架旋转变成了函数递归

```
半径(10000倍输入)：#3d.pentagram_epi.r int

摆线进行圆周运动时的转速：#3d.pentagram_epi.speed int
范围[1,7200000]。参考值：输入20000适中
函数递归的次数与转速相关，为了确保绘制出完整的图形，转速越慢得到的粒子坐标越多，转速越快粒子坐标越少。递归次数上限=7200000/转速

五角星的横滚角(10000倍输入)：#3d.pentagram_epi.roll.θ int

计算坐标：
execute as b09e-44-fded-6-efa5ffffef64 run function
large_number:particle/3d_ar_rotation_pentagram/epicycloid/start
```

图形显示

```
输出相对坐标列表：
storage large_number:math 3d_ar_rotation_pentagram_pos
其中每一个子列表的第一项是x，第二项是y

显示粒子：
execute positioned x y z rotated x y run function
large_number:particle/3d_ar_rotation_pentagram/particle
执行朝向就是五角星的朝向，执行位置就是五角星的位置
```

♦ 三维空间任意方向的椭圆

```
1000倍输入 a：#3d.ellipse.a int
1000倍输入 b：#3d.ellipse.b int
1000倍放大后的粒子圆的输入区间为[1,2147483]

粒子密度：#3d.ellipse.density int
粒子密度就是每隔"n/10"度描一个点，范围为1~3600

横滚角(1000倍输入)：#3d.ellipse.roll.θ int
```

```
计算坐标：
execute as b09e-44-fded-6-efa5ffffef64 run function
large_number:particle/3d_ar_ellipse/start

输出相对坐标列表：storage large_number:math 3d_ar_ellipse_pos
其中每一个子列表的第一项是x，第二项是y

显示粒子：
execute positioned x y z rotated x y run function
large_number:particle/3d_ar_ellipse/particle
执行朝向就是椭圆的朝向，执行位置就是椭圆的位置
```

♦ 粒子球 (斐波那契网格)

球面均匀取点方法：若是从球面上取n个点，则是把球横着切成n层，让这些点沿着球面从球底爬到球顶，每爬一层就绕着这一层的圆心转0.618圈。

相关链接：https://zhuanlan.zhihu.com/p/25988652

```
球的半径：storage large_number:math 3d_hsphere_pos_R 0.0

在球面上取的点的数量：#3d.hsphere.points int
输入区间为[1,40000]

计算坐标：
execute as b09e-44-fded-6-efa5ffffef64 run function
large_number:particle/3d_hsphere/start

输出相对坐标列表：storage large_number:math 3d_hsphere_pos
其中每一个子列表的第一项是x，第二项是y

显示粒子：
execute positioned x y z rotated x y run function
large_number:particle/3d_hsphere/particle/start
传入执行位置和执行朝向
```

另可在球面上的点上执行其他命令：

```
在球面上的点上要执行的命令：
storage large_number:math 3d_block_hsphere_execute "setblock ~ ~ ~ glass"

执行命令：
execute positioned x y z rotated x y run function
large_number:particle/3d_block_hsphere/set/start
传入执行位置和执行朝向
```

♦ 全息粒子投影 - 16x16x16投影至1x1x1

把染色混凝土投影为dust粒子

扫描一次后，粒子颜色和坐标等信息会存入数据库，就算扫描区清空了也一样可以投影

添加可解析方块：
在函数 "particle/holographic_projection/if" 里的第18行开始添加如下格式的命令：
`execute if block ~ ~ ~ <方块ID|方块标签>[方块状态]{数据标签} run data modify storage large_number:math temp_particle set value "<dust粒子的四个特殊参数>"`
方块状态和数据标签都是可选的

```
先扫描：execute positioned x y z run function
large_number:particle/holographic_projection/scan.start
执行位置需要在扫描区域的西北下角
聊天栏出现"全息粒子投影：扫描完成！"时即为扫描完成。

投影：execute rotated 0.0 0.0 positioned x y z run function
large_number:particle/holographic_projection/execute with storage
large_number:math holographic_projection_database
需要传入投影点和投影角度，投影的位移和旋转的基点在投影的底面中心
会触发函数宏的缓存机制，可高频执行

清空数据库：data remove storage large_number:math holographic_projection_database
```

♦ 抛物线

1.把三点坐标解析为二次函数表达式的abc：large_number:parabola/3point_ope_coef.abc

原理：加减消元法求解三点对应的三元一次方程组。

```
输入：storage large_number:math parabola_points [[0.0,0.0],[0.0,0.0],[0.0,0.0]]
输入二维坐标点，取整数和第一位小数

输出(放大一千倍)：#coef.a int，#coef.b int，#coef.c int
```

2.解析二次函数的表达式为点的相对坐标：large_number:parabola/analysis.start

公式：`f(x)=ax²+bx+c`

解析后坐标会存入列表里，不用每次都解析

```
以一千倍输入系数：#coef.a int，#coef.b int，#coef.c int
以一百倍输入起始X值：#parabola_expre_x.start int
输入步数：#parabola_expre_x.length int
以100倍输入步长：#parabola_expre_x.step_size int

输出相对坐标列表：
x：storage large_number:math parabola_expre_x
y：storage large_number:math parabola_expre_y

显示抛物线表达式：set #parabola_.tellraw int 1
显示以下JSON文本便可显示抛物线表达式：
["f(x)=",{"nbt":"parabola_tellraw.a","storage":"large_number:math"},"x²",
{"nbt":"parabola_tellraw.1","storage":"large_number:math"},
{"nbt":"parabola_tellraw.b","storage":"large_number:math"},"x",
{"nbt":"parabola_tellraw.2","storage":"large_number:math"},
{"nbt":"parabola_tellraw.c","storage":"large_number:math"}]
```

3.显示抛物线的轨迹：`execute positioned x y z rotated 0.0 0.0 run function large_number:parabola/particle`

需要传入执行位置和执行朝向

```
模式：#parabola_expr_particl_mode int
可选1或2，区别就是粒子的参考系不同，可以应对不同的旋转需求
模式1粒子是从执行朝向的左方向出发，模式2是粒子从执行朝向的前方出发

抛物线的位移和旋转基点是它的起始点
```

♦ 阿基米德螺线 (等速螺线)

公式：`r=a+bθ`

```
1000倍输入a：#archimedean_spiral.a int
1000倍输入b：#archimedean_spiral.b int
100倍输入起始角度：#archimedean_spiral.startθ int
100倍输入弧长步长：#archimedean_spiral.arc_size int
100倍输入角度步长：#archimedean_spiral.θ_size int
步数：#archimedean_spiral.length int
n步后使用弧长来计算点的间隔：#archimedean_spiral.to_arc int

计算坐标：function large_number:particle/archimedean_spiral/start

输出相对坐标列表：
x：storage large_number:math archimedean_spiral_out_listX
y：storage large_number:math archimedean_spiral_out_listY

显示粒子：execute positioned x y z rotated x y run function
large_number:particle/archimedean_spiral/particle
需要传入执行位置和执行朝向
```

一个较好的预设：a为100，b为8，起始角度为0，弧度步长35，角度步长1000，30步后使用弧长

♦ 等角螺线 (对数螺线)

公式：`θ=a*ln(b*r)`

```
载入初始数据库：function large_number:ln/ln_database

a：#equiangular_spiral.a int
b：#equiangular_spiral.b int
1000倍输入起始半径：#equiangular_spiral.start_r int
1000倍输入半径步长：#equiangular_spiral.r_size int
步数：#equiangular_spiral.length int

计算坐标：function large_number:particle/equiangular_spiral/start

输出相对坐标列表：
x：storage large_number:math equiangular_spiral_out_listX
y：storage large_number:math equiangular_spiral_out_listY
```

```
显示粒子：execute positioned x y z rotated x y run function
large_number:particle/equiangular_spiral/particle
需要传入执行位置和执行朝向
```

一个范例：a为5000，b为560，起始半径是0，步长是50，步数是250

♦ 二维网格排列

```
1000倍输入 行间隔：#Matrix_arrangement.rsize int
1000倍输入 列间隔：#Matrix_arrangement.csize int
1000倍输入 偶数行偏移：#Matrix_arrangement.tab int
行数：#Matrix_arrangement.Rows int
列数：#Matrix_arrangement.Columns int
计算坐标：execute as b09e-44-fded-6-efa5ffffef64 run function
large_number:matrix_arrangement/start

输出的是阵列的x和z的相对坐标列表
坐标的相对x值列表：storage large_number:math matrix_arrangement_X
坐标的相对z值列表：storage large_number:math matrix_arrangement_Z

一个使用函数宏访问坐标列表的范例：execute positioned x y z rotated x y run function
large_number:matrix_arrangement/summon
需要传入执行位置和执行朝向
```

♦ 二阶贝塞尔曲线

公式：

$$B\left(t\right)=\left(1-t\right)^{2}P_{0}+2t\left(1-t\right)P_{1}+t^{2}P_{2},\ t\in\left[0,1\right]$$

```
输入

一千倍输入三点坐标：
#bezier_curve_II.P0.x int
#bezier_curve_II.P0.y int
#bezier_curve_II.P0.z int

#bezier_curve_II.P1.x int
#bezier_curve_II.P1.y int
#bezier_curve_II.P1.z int

#bezier_curve_II.P2.x int
#bezier_curve_II.P2.y int
#bezier_curve_II.P2.z int

一万倍输入t的步长：#bezier_curve_II.t.size int

计算坐标：function large_number:particle/bezier_curve_2/start

输出相对坐标列表：
x：storage large_number:math bezier_curve_II_list_X
```

```
y：storage large_number:math bezier_curve_II_list_Y
z：storage large_number:math bezier_curve_II_list_Z

显示粒子：execute positioned x y z rotated x y run function
large_number:particle/bezier_curve_2/particle
传入执行位置和执行朝向
```

♦ 心形线

公式：

$$\text{上半段：}\sqrt{r\left|x\right|-x^2}$$
$$\text{下半段：}\frac{r}{2}\left(\arccos\left(1-\left|\frac{2x}{r}\right|\right)-\pi\right)$$

```
半径 (10000倍输入)：#heart-shaped_line.r int
上半段粒子密度 (单位为角度度数，100倍输入)：#heart-shaped_line.t_d int
下半段粒子密度 (单位为格，10000倍输入)：#heart-shaped_line.t int

在心形线的断开处描点来把图像连起来 (描点次数)：#heart-shaped_line.extra int
修复图像的描点宽度 (单位为格，10000倍输入)：#heart-shaped_line.t_x int
心形线上下两段没连起来是由于计算误差造成的

计算坐标：
execute as b09e-44-fded-6-efa5ffffef64 run function large_number:particle/heart-shaped_line/start

输出相对坐标列表：storage large_number:math heart-shaped_line_Pos
其中每一个子列表的第一项是x，第二项是y

显示粒子：execute positioned x y z rotated x y run function
large_number:particle/heart-shaped_line/particle
传入执行位置和执行朝向
```

♦ 色轮

```
显示色环：execute positioned x y z rotated x y run function
large_number:color_wheel/particle1
传入执行位置和执行朝向

10000倍输入色环上的颜色指针角度(逆时针方向)：#color_wheel.angle.input int
输入区间为：[0,3600000]
用粒子标记色环指针指向的位置：set #color_wheel.see_marker int 1
计算色相立方的颜色信息：function large_number:color_wheel/in/start

输出色环指针处的RGB值：
#color_wheel.output.R int
#color_wheel.output.G int
#color_wheel.output.B int
色相立方的粒子信息表：storage large_number:math color_wheel_color_cube_RGB
```

```
显示色相立方：execute positioned x y z rotated x y run function
large_number:color_wheel/in/particle/start
传入执行位置和执行朝向

10000倍输入色相立方的颜色坐标：
#color_cube.u int
#color_cube.v int
这是两个百分比，u表示颜色坐标和色相立方右上角起始点的横向距离，v表示颜色坐标和色相立方右上角起始点的纵向距离
输入区间皆为[0,10000]

计算颜色坐标：function large_number:color_wheel/in/ope_uv_color/start

输出RGB值：
#color_cube.R int
#color_cube.G int
#color_cube.B int
```

♦ 直线

```
1000倍输入 总长度：#3d_straight_line.length int
1000倍输入 点的间隔：#3d_straight_line.density int

计算坐标：function large_number:particle/3d_straight_line/start

输出相对坐标列表：storage large_number:math 3d_straight_line_Pos
直线是一维图形，所以只有一个变量

显示粒子：execute positioned x y z rotated x y run function
large_number:particle/3d_straight_line/particle
传入执行位置和执行朝向
```

♦ 粒子正多边形

```
1000倍输入 图形的横滚角：#regular_polygon.startθ int
当角度为-90时，图形的第一个顶点是垂直向上的
1000倍输入 图形的半径：#regular_polygon.r int
1000倍输入 粒子的间隔：#regular_polygon.size int
图形的边数：#regular_polygon.n int

计算坐标：
内接正多边形：execute as b09e-44-fded-6-efa5ffffef64 positioned .0 .0 .0 run
function large_number:particle/regular_polygon/start
外切正多边形：execute as b09e-44-fded-6-efa5ffffef64 positioned .0 .0 .0 run
function large_number:particle/regular_polygon/tangent_start

输出相对坐标列表：storage large_number:math regular_polygon_Pos
其中每一个一级子列表表示多边形的一条边，每个二级子列表的第一项是x，第二项是y

显示粒子：execute positioned x y z rotated x y run function
large_number:particle/regular_polygon/particle
```

```
传入执行位置和执行朝向
```

♦ 行列式

1.判断输入值是否为行列式：large_number:determinant/order

行列式输入规则：必须有两层列表，每个子列表表示一行。如果该行某个元素为0也必须输入0，不支持元素省略。

例如 [[4,15,7],[6,13,4],[28,2,12]] =

$$\begin{vmatrix} 4 & 15 & 7 \\ 6 & 13 & 4 \\ 28 & 2 & 12 \end{vmatrix}$$

```
输入：storage large_number:math determinant_evaluate.input [[4,15,7],[6,13,4],
[28,2,12]]

阶数：#determinant.order int
-1表示输入的行列式错误
```

2.基础行列式求值：large_number:determinant/evaluate/start

仅支持1~7阶，输入值仅接受int

用代数余子式一层层按行展开，最终把高阶行列式展开成多个三阶行列式

```
输入：storage large_number:math determinant_evaluate.input [[4,15,7],[6,13,4],
[28,2,12]]

输出：storage large_number:math determinant_evaluate.output
阶数：storage large_number:math determinant_evaluate.order
```

3.整数列表的逆序数：large_number:determinant/inversion_number/start

规定正序排列为从小到大

```
输入：storage large_number:math invers_num_inp [0,1,7,9,6,14,28,5]

输出：#invers_num.output int
若输入的列表没有重复项，且逆序数=(元素数-1)*元素数/2，则列表元素为从大到小排列。
```

♦ 参考文献：

> 小豆数学库：https://github.com/xiaodou8593/math2.0
> 知乎.手动开根——牛顿迭代法：https://zhuanlan.zhihu.com/p/497508702
> 知乎.手动开根——竖式开方法：https://zhuanlan.zhihu.com/p/517358606
> 小豆.用命令做一个简易的开根号：https://www.bilibili.com/read/cv5789989
> 天起源.T算法库：https://www.mcmod.cn/class/11569.html
> 计算机系统数学原理：http://mathmu.github.io/publications/mathematical-theory-of-computer-algebra-system
> 【动画密码学】Base64编码&解码算法：https://www.bilibili.com/video/BV1Hp4y1g7Ex

> 卡儿.实数平方根的估值与连分数展开 (提取码 sr8j)：https://pan.baidu.com/s/1eoeChhk7xukIIYxexmMwJQ?pwd=sr8j
> 知乎.最大公约数GCD算法：https://zhuanlan.zhihu.com/p/38100838
> 卡儿.《我的世界》【1.16.5】Java版实用粒子教程：https://www.bilibili.com/read/readlist/rl651851
> 数值分析 第五版 (李庆扬 王能超 易大义) (提取码：dker)：https://pan.baidu.com/s/17aYm5onfSbsxH4TmL00mmQ?pwd=dker

工具：GeoGebra，Desmos，Excel，Python