# 几个数学模板

*duangsuse*
22, Jul 2019

## 要命的数学 | F**king Math

今天谈几个数学相关的问题。
首先我们看看某 OI 题目：

$$resolve(n, m) = \sum_{i=1}^{n}\sum_{j=1}^{m} ij(n \mod i)(m \mod j) \mod 10^9 + 7$$

where

$$n \geq 10 \wedge m \leq 10^9$$

输入输出格式这种 trivial 的问题我们就不管了。输入的是一组参数数据、输出结果。

还得在 1s 内解出来，不然还不行（ICPC 赛制，答案正确 TLE 还没分）
给的复杂度要求是 $o(\sqrt{n})$
然后可以给个例子：

$$resolve(10, 10) = 6561$$

我这个半工程系半理论系的看到首先以为是翻译题，没有注意到时间限制和数值稳定问题

> Young Zy, [21.07.19 17:33]
> > 完全自闭
> > 然后我因为取余写自闭了，至少白给了8发
>
> duang suz, [21.07.19 18:53]
> [In reply to Young Zy]
> < 卡了你两个小时，本来已经通过了，可是你为了优化 10x 就又多花了很多时间？...
>
> duang suz, [21.07.19 18:56]
> [In reply to Young Zy]
> ```
> def resolve(n, m)
> ac = 0
> (1..n).each do |i|
>   (1..m).each do  |j|
>     ac += i*j*(n % i)*(m % j)
>   end
> end
> return ac % (10**9 + 7)
> end
> ```
>
> （也可以 Map & Sum, Kotlin 里一般这么做大概，当然这都应该看优化了的说，比如并行计算和代数化简）
> 我还以为是高大上算法题呢... 数学公式翻译啊... 连列数学公式都不需要自己做，这降低了好大难度档次的说...
>
> 开始看这 MathJax 的排版我还以为是逆天级别的题目... 数学 + 算法...
>
> [In reply to Young Zy]
> < 又不是位运算优化取余或者自己实现整除和取余(divmod) (
> < def fmod(n, m): return (n|m)-n
> < 话说那个『快速』取余是怎么定义来着...
>
> [In reply to Young Zy]
> < 是数值安全（防溢出什么的）的题？
>
> [19:12 21.07.19] ,ƍɒmƨuoY ɈႱɘɔɿuF
> > 多久算出来？
> > 要速度的
>
> [In reply to Young Zy]

> < 我还以为会溢出（
>
> [20:22 21.07.19] ,ꝯꭤꭥꝸꭒoY ɫ1VƐꞁɯꟻ
> > 会
> > 绝对会
> > 多个错误撞一起
> > 看哪个先碰到
>
> duang suz, [21.07.19 20:22]
> < 等价变换啊，很多嵌入式算法都是这么解决的

## 我考虑了一会后想到 | What I Thought

因为有这个题目，我出门散步的时候连歌都没有听好，一直在想 $\sum_{accum=initial}^{max}$ operator 变换和 $\mod$ 操作符定义的问题…
而之前我好像连乘方 $x^2$ 和 $\log_2$、$\sqrt[3]{x}$ 都分不清…
走路的时候我想到了一种『优化』可能：

$$\forall a, b, a_1, a_2, \cdots, a_n . a_1 = a_2 = \cdots = a_n \Rightarrow [a_1, a_2, \cdots, a_n] \mod b = a \mod b$$

（其中 $a_0, \cdots, a_n$ 是从 $a$ 中截取到相等的部分）
（开始的时候思路蛮不清晰的，式子还写错了，我整理了一下）
但是早在我想到的时候就被推翻了…

$$100 \mod 10 \neq \overbrace{[10, \cdots, 10]}^{10} \mod 10$$

但是我依然怀疑这个式子是正确的，于是又想是不是

$$a \mod b = \overbrace{[a_1, a_2, \cdots, a_n]}^{n} \mod b + n$$

但是又瞬间被打脸了：

$$100 \mod 1 \neq \overbrace{[10, \cdots, 10]}^{10} \mod 1 + 10$$

> duang suz, [21.07.19 20:23]
> 我走路的时候想了一会，否定了以下的脑残假设，打算过会整理一下成文，算作进步… 虽然我还是不会
>
> （此处省略若干行）
>
> 我…
>
> 数值的我直觉比较差… 因为完全模拟做不到、一些别人都已知道的代数套用我又不会记，然后我整理的速度也比别人慢很多…

$$\sum_{i=1}^{n}\sum_{j=1}^{m} ij(n \mod i)(m \mod j) mod(10^9 + 7)$$

之前排版过相同的题目式子

> duang suz, [21.07.19 20:38]
> [In reply to Young Zy]
> < 🤔 我要手动展开很久… 你是怎么解的？是代数化简吗？
>
> [20:39 21.07.19] ,ꝯꭤꭥꝸꭒoY ɫ1VƐꞁɯꟻ
> > 你知道同余定理嘛
>
> duang suz, [21.07.19 20:39]
> [In reply to ꝯꭤꭥꝸꭒoY ɫ1VƐꞁɯꟻ]
> < 不知道，恐怕我得手动推…

然后我就去 Haskell haddoc base:v:mod 那里找了一下 `div` 的定义，找不到（估计是 Intrinsic），但是找到了一个性质

$$(a \div b \times b) + a \mod b = a$$

“甚至我连 mod 的定义都得再确认一遍，如果除数大于被除数应该怎么办，现在看来... 10/100=0; 10 mod 100 = 100”
结果我正在准备分析的时候，直接发答案了

> Young Zy, [21.07.19 20:54]
> > 官方的题解出来了，我截个图给你们
> > 反正是个数学题

### 题解 | "Immediate" Solution

$$\sum_i^n \sum_j^m ij(n \mod i)(m \mod j) \mod 10^9+7$$
$$= \sum_i^n i(n \mod i) \sum_j^m j(m \mod j)$$

这是完全一致，所以 $\mod 10^9+7$ 根本就是障眼法，而且 $\mod$ ~~有分配律？~~
$ij$ 都容易计算（等差求和公式修改一下），可是 $\mod 10^9+7$ 难道是永远 $\frac{x}{10^9+7}=0$ ？为什么？这个计算怎么就等价？常量 $10^9+7$ 是怎么来的？
所谓的两个 $\sum$ 『独立』是怎么个『独立』法？为什么 $\sum_i^n\sum_j^m ija = \sum_i^n ia\sum_j^m ja$ ？（也可能和后面的 $\mod 10^9+7$ 有关）这些事情必须知道
这些事情我不擅长（因为我有点讨厌变形），所以这里不能说（
我编程的方法和数学有相通之处，但我又不是学数学的人

我从小数学不好的原因，一方面是我自己不能理清思维，并且想的很天真（比如 $1+1$ 就是 $2$、$2*2$ 就是 $4$，我不知道 $4\equiv(2*2)\equiv2+2$，这也是后来我连完全平方公式都只能勉强强记强行套用的缘由），另一方面是很多数学老师太 implicit 了，他们就是只告诉你『该怎么做』，就是不把最细致的过程写明白，因为在他们眼里那些东西都是极其 immediate 的，当然那对聪明的学生不算什么，他们很轻易地就能立即 get 到老师的 point，迅速进入思考问题本身的模式，而我偏偏是个笨学生，被拦在几个看不懂的基本组合之外永远不得要领。

我觉得有时候数学比起计算机科学来说更不够亲民的锅 *更可能是在做学问的人身上*，数学家对『简洁』的崇尚使得它对那些没有那么强隐式推导能力的人来说，太反直觉了，即便是数学基础教育，那些『看不懂』的孩子就没有人跟他们去解释『为什么』，所有教程也都只是为大部分人准备的，这些资料不会告诉你那些简单却是最基础构造单元的东西，只是在你已经（或许是隐约地）感受到他们后直接开始暴力训练

按天资去选择啊。

余下的部分主要放一些简单的相关内容

### sqrt, log 和乘方函数 | Power Operators

乘方是乘法的组合，

$$x^n = x\overbrace{\times x \times x \times \cdots \times x}^{n}$$

其中 $x$ 为『底数』、$n$ 为『指数』（重复次数），实例比如

$$2^1 = 2 = 2$$
$$x^2 = xx = x \times x$$
$$3^3 = 3 \cdot 3 \cdot 3 = 3 \times 3 \times 3 = 9$$

有时候认为是乘法，其实更多时候都是『直接并列』而已。只是因为数学并列两个数值变量可以作为乘(times) 的简记法，所以可以这样。（以『+』并列则可以做加法）

有一个问题：除了这样，$n$ 个 $x$ 相乘还能做什么计算？

开方 | Roots

开方是给定最终表示、并列次数求并列项

$$(8 \times 8) \equiv 64 \Rightarrow \sqrt[2]{64} = 8$$

$$\sqrt[n]{x^n} = x$$

$$\sqrt[n]{(\overbrace{x \cdot x \cdots x}^{n})} = x$$

牛顿开方法就是把每个可能的乘法因子尝试一遍，逼近求解

对数 | Logarithm

对数是给定最终表示、并列项求并且次数

$$(2^8) \equiv 256 \Rightarrow \log_2 256 = 8$$

$$\log_x(x^n) = n$$

$$\log_x(\overbrace{x \cdot x \cdots x}^{n}) = n$$

利用定义，也可以进行变换得到派生性质，这是转化的基础。

$$\forall nx.\, k = \log_n x \Rightarrow x = n^k$$

$$x = 64, n = 8 \Rightarrow k = \log_8 64 = 2; 64 = 8^k = 8^2$$

## divrem 的『图示』 | Property Definition for div & rem

- 数是什么？这里的数是自然数 (Natural Numbers)

$$Nat = \begin{cases} 0 \in Nat \\ x \in Nat \Rightarrow (x+1) \in Nat \end{cases}$$

$$\begin{aligned} zero &= 0 \\ succ(x \in Nat) &= x + 1 \\ pred(x \in Nat) &= x - 1 \\ pred(zero) &= undefined \\ \forall x.\, (x+1) - 1 &= x \end{aligned}$$

由 $\forall x.\, (x+1) - 1 = x$ 可以推出 $\forall x \in Nat.\, (x-1) \in Nat$

$$\begin{array}{rcll} (0+1) \in Nat & \Rightarrow & ((0+1)-1) \equiv 0 & \in Nat \\ \forall (x+1) \in Nat & . & ((x+1)-1) \equiv x & \in Nat \\ & & pred(succ(0)) \equiv 0 & \in Nat \\ \forall succ(x) & . & pred(succ(x)) \equiv x & \in Nat \end{array}$$

▼ Haskell

```haskell
-- data Nat = O | S Nat
data Nat where
  O :: Nat
  S :: Nat -> Nat
zero = O :: Nat
succ = S :: Nat -> Nat
pred (S x) = x
pred O = undefined
-- (succ . pred) = (S . \(S x) -> x) = id
```

- 加法是什么？

$$add(a,b) = \begin{cases} b, & a = zero \\ add(pred(a), succ(b)), & otherwise \end{cases}$$

In other words，把 $a$ 的每个 $succ$ 转移到 $b$ 上

$$\begin{aligned} add(succ(x),b) &= add(x, succ(b)) \\ add(zero,b) &= b \end{aligned}$$

▼ Haskell

```haskell
add :: (Nat, Nat) -> Nat
add (a, b)
  | a == zero = b
  | otherwise = add (pred a, succ b)

add' :: (Nat, Nat) -> Nat
add' (S x, b) = add' (x, S b)
add' O b = b
```

- 减法是什么？

$$sub(b,a) = \begin{cases} b, & a = zero \\ zero, & b = a = zero \\ undefined, & b = zero \\ sub(pred(b), pred(a)), & otherwise \end{cases}$$

换句话说，把每个 $pred$ 转移到 $b$ 上

$$\begin{aligned} sub(b, \quad zero) &= b \\ sub(succ(b'),succ(a')) &= sub(b', a') \\ sub(zero, \quad zero) &= zero \\ sub(zero, \quad a) &= undefined \end{aligned}$$

▼ Haskell

```haskell
sub :: (Nat, Nat) -> Nat
sub (b, a)
  | b == O & a /= O = undefined -- 负数不是自然数
  | a == O = b
  | otherwise = sub (pred b, pred a)

sub' :: (Nat, Nat) -> Nat
sub (b, O) = b
sub (O, S _) = undefined
sub (S b', S a') = sub (b', a')
```

- 乘法是什么？

$$a \times b = \overbrace{a + a + \cdots + a}^{b}$$

$$times(a,b,n=0) = \begin{cases} n, & a = 0 \\ times(pred(a), b, add(b,n)), & otherwise \end{cases}$$

▼ Haskell

```haskell
times :: (Nat, Nat) -> Nat
times a b = times' a b O
  where
    times' O _ n = n
    times' (S a') b n = times' a' b (add (b, n))
```

```
times' :: (Nat, Nat) -> Nat
times' a b = foldl (\ac, x -> add (x, ac)) O (take b (repeat a))

foldn :: (Nat -> Nat -> Nat) -> Nat -> (Nat -> Nat)
foldn f v O = v
foldn f v x = let (S x') = x in foldn f (f v x) x'

times'' = foldn (curry add) O
```

- 求幂（乘方）是什么？写累了，说起来，其实即便有交换律存在，顺序（主语和宾语）很重要，可惜我级别太低无力维护了。

$$a^b = \overbrace{a \times a \times \cdots \times a}^{b}$$

▼ Haskell

```
power a b = foldn ((curry times) a) (S O) b
```

- 除法是什么？$b$ 里面有几个 $a$

$$div(b, a, n = 0) = \begin{cases} div(sub(b, a), a, succ(n)), & gt(b, a) \vee a \equiv b \\ n, & otherwise \end{cases}$$

▼ Haskell

```
div :: (Nat, Nat) -> Nat
div (b, a) = div' (b, a, O)
  where div' (b, a, n)
    | eq(b, a) or gt(b, a) = div(sub(b, a), a, succ(n))
    | otherwise = n
```

- 取余法是什么？神奇海螺：为啥不用 divrem 呢？

$$rem(b, a, n = 0) = \begin{cases} rem(rem(b, a), a, succ(n)), & gt(b, a) \vee a \equiv b \\ b, & otherwise \end{cases}$$

从取余数也可以推出一个简单的性质：当除数大于被除数，余数等于被除数、商等于零

$$\neg gt(b, a) \Rightarrow rem(b, a) \equiv b \wedge div(b, a) \equiv 0$$

$$b < a \Rightarrow rem(b, a) \equiv b \wedge div(b, a) \equiv 0$$

所以取余操作符的性质等式才能成立：

$$\forall ba.\, a \times \llcorner \frac{b}{a} \lrcorner + rem(b, a) = b$$

▼ Haskell

```
rem :: (Nat, Nat) -> Nat
rem (b, a) = rem' (b, a, O)
  where rem' (b, a, n)
    | eq(b, a) or gt(b, a) = rem(sub(b, a), a, succ(n))
    | otherwise = b

divrem :: (Nat, Nat) -> (Nat, Nat)
divrem (b, a) = rem' (b, a, (O, O))
  where divrem' (b, a, (dn, rn))
    | eq(b, a) or gt(b, a) = divrem(sub(b, a), a, (succ(n), O))
    | otherwise = (dn, b)
```

- 比较法是什么？包含相等性和『小于』lessThan

$$eq(a,b) = \begin{cases} true, & a = b = 0 \\ eq(pred(a), pred(b)), & a \neq 0 \wedge b \neq 0 \\ false, & otherwise \end{cases}$$

$$gt(b,a) = \begin{cases} true, & a = 0 \wedge b \neq 0 \\ false, & b = 0 \wedge a \neq 0 \\ false, & a = b = 0 \\ gt(pred(b), pred(a)), & otherwise \end{cases}$$

$$lessThan(a,b) = gt(b,a)$$

▼ Haskell

```haskell
eq :: (Nat, Nat) -> Bool
eq (O, O) = True
eq (S a', S b') = eq (a', b')
eq (_, _) = False

gt :: (Nat, Nat) -> Bool
gt (O, S _) = False
gt (S _, O) = True
gt (O, O) = False
gt (S b', S a') = gt (b', a')

lessThan :: Nat -> Nat -> Bool
lessThan = flip (curry gt)
```

- 被除数是什么？最后一问，实在没有力气写了，我相信我是这个问题写的最仔细的人了可能

$$\underbrace{\overbrace{a + a + \cdots + a}^{\frac{b}{a}} + \overbrace{c}^{b - a \times \frac{b}{a}}}_{b}$$

然后你可能要问了，那 $b - a \times \frac{b}{a}$ 不是可以 约分嘛，这不 $c \equiv b - b \equiv 0$

$$\underbrace{\overbrace{a + a + \cdots + a}^{\frac{b}{a}}}_{b}$$

所以算 $c$ 要整除 ( Haskell 里 b \`div\` a 是整除 (/) b a 是实除 )
~~( 虽然引入小数后除不尽就不可能了~~

$$\underbrace{\overbrace{a + a + \cdots + a}^{\llcorner \frac{b}{a} \lrcorner} + \overbrace{c}^{b - a \times \llcorner \frac{b}{a} \lrcorner}}_{b}$$

定义是递归的，所以不好理解 ( 应该说这是对 $\frac{b}{a}$ 的定义，可是 $\llcorner \frac{b}{a} \lrcorner$ 实际应该被… 认为是先除法再取整的操作，而不是取整和除法定义的 结合，这是『模式』化可能造成的一个混淆 )

▼ Haskell

```haskell
{- 不好意思，我已经不知道如何使用 Haskell 表达它了 -}
Integral :: * -> Constraint
div :: Integral a => a -> a -> a
(/) :: Fractional a => a -> a -> a
instance Integral Int -- Defined in 'GHC.Real'
```

- 如何和普通的数字进行转换？不过 Nat 和 Int 并不是同构的
  请读者自己开动脑筋，用 Haskell 完成 fromInt 和 toInt 函数的定义

## 从等式分析 ( 变换 ) 看 sum ( 求和公式 ) | The $\sum$ Operator

学到了很多（虽然都只是加强理解而已），这很好，但是以后这种问题在编程中还可能遇到很多次，比这更难的问题也可能遇到很多次。
既然这次遇到了，就解决它，不要下次摸瞎。

$$resolve(n \in (10, \infty), m \in (0, 10^9)) = \sum_{i=1}^{n} \sum_{j=1}^{m} ij(n \mod i)(m \mod j) \mod 10^9 + 7$$

看起来好像是铁石一样坚固，并且无法有任何的拆分变换，其实也可能是理解还不够深，首先从最简单的例子开始。

$$\sum_{i=0}^{100} i$$

先尝试计算一下它，进行定义展开（也可以叫施用）：

$$\sum_{i=a}^{n} body = joinBy(+)(let \square i = a' \in (a, n) in \square body)$$

其实这个定义模板没有多大用，因为大家基本都看不出什么…

$$\sum_{i=0}^{100} i = 0 + 1 + 2 + \cdots + 100 = (0 + 100) + (1 + 99) + (2 + 98) + \cdots + (50 + 50) = \frac{(0 + 100) \cdot 100}{2} = 5000$$

呃… 等差求和公式是『首项+末项*项数/2』，大概就是 zip rev 一下，从每项差 1 的推广下去吧，本身属于数学的一种优化变形方案
据说是某位西方国家的天才少年弄出来的，在那之前都在用傻方法…
你看它会每次从 $xs$ 和 $reversexs$ 取出一项操作，直到（inclusive，包含此 case）取出的两项相等

$$\frac{(first + last) * size}{2}$$

我们现在先推广一下这个公式，让它能适合每项偏差 $d$ 的序列，那就是在设计一个算法了，一般来说，像我是不得不列一个表格来找规律的
据说大佬都是直接从属性和定义推，列表的是那种不需要草稿纸的列（

如果 $first = 1; last = 100;$
$d = 2$ 如何

| $i$ | $A_{i+1}$ | $A_i + (A_{i+1})$ |
|---|---|---|
| 1 | 3 | 4 |
| 3 | 5 | 8 |
| 5 | 7 | 12 |
| 7 | 9 | 16 |
| 9 | 11 | 20 |

看来即使 $d \neq 1$ 也是比较有规律，看看原公式是否可以直接推广

如果 $first = 1; last = 100;$
$d = 2$ 如何

| $i$ | $A_{n-i}$ | $A_i + (A_{n-i})$ |
|---|---|---|
| 1 | 99 | 100 |
| 3 | 97 | 100 |
| 5 | 95 | 100 |
| 7 | 93 | 100 |
| 9 | 91 | 100 |

看起来好像的确从 $A_i$ 和 $A_{n-i}$ 数起来，$i$ 递增的话前者会递减 $d$、后者递增 $d$，所以相同的优化思路（应该是）在所有 $x \in \mathbb{N}$ 都有效，那么我们就尝试泛化出公式里的『$1$』，推广差为 $1$ 的等差求和操作为差为 $n$ 的：
先考虑一下公式 $\frac{(first+last)*len}{2}$ 优化的部分：它合并了对称项目，合并了 (+) 为乘，要修改的部分应该是『求合并的项目个数』部分
$size \equiv (last - first)$

区别在于，步长会导致要求和的项目发生变化，怎么反映这个变化呢？

```python
for i in range(1, 11): print(str(i) + " " + str(len(list(range(1, 101, i)))))
```

| d | card | 100 mod d |
| --- | --- | --- |
| 1 | 100 | 0 |
| *2 | 50 | 0 |
| 3 | 34 | 1 |
| *4 | 25 | 0 |
| 5 | 20 | 0 |
| *6 | 17 | 4 |
| 7 | 15 | 2 |
| *8 | 13 | 4 |
| 9 | 12 | 1 |
| *10 | 10 | 0 |

以上在 $i$ 是偶数的时候都打了星，总感觉可以有递归…
可是之后，我才发现，其实我们是要一个函数 $icount(n, d) = \cdots$
然后这个函数呢？我忽然发现它不就是整除嘛…

$$\underbrace{\overbrace{a + a + \cdots + a}^{\lfloor \frac{b}{a} \rfloor} + \overbrace{c}^{b - a \times \lfloor \frac{b}{a} \rfloor}}_{b}$$

我们这里要拿到的不就是 $a$ 的个数么… 所以最终的定义为：

$$\frac{(a+b) \times \lfloor \frac{(a+b)}{d} \rfloor}{2}$$

最后再看看这个：

$$fun = \sum_{i}^{n} \sum_{j}^{m} i \cdot j$$

知道 join 是有规律的，那就找规律

$$n = 10; m = 11 \Rightarrow fun = (1 \times 1, \cdots, 1 \times 11) + (2 \times 1, \cdots, 2 \times 11) + \cdots + (10 \times 1, \cdots, 10 \times 11)$$

那可以试着泛化一下，就是矩阵运算（没意思），不过，可以有这种变换：（

## 完了 | Aborted

目前我做事情是有时间限制的，超过了我就会想办法尽快结束。
目前写这篇文章已经花了我一天的时间了… 即时我还有递归的 C++ 二元操作链条解析器要写…
而且写作的过程中即使有 gnome-break-timer 在催我也很久没有休息的… 总之这次先到这里了

Time limit exceeded
*Process finished. Your writing aborted with exit code -1*

显示 NSFW 内容 ?