类比思想在解题中的应用

【关键字】 思想；类比；相似性；对应。

【摘要】：

类比，是一种试图建立未知的问题与已知的问题之间的联系，从而利用已知的解题方法去解决新的问题的思路。本文首先通过分析具体的例子，指出类比解题不仅仅是注意到了表面上的相似性，更是建立了已知问题和未知问题之间的对应关系。然后，本文将通过另一个例子，论述表面相似性与内在的对应之间的关系，并且指出利用类比解题的过程是从表面相似性上升到一一对应的过程。

引：解题，从熟悉的地方开始

面对一个新的问题应该如何着手去分析解决呢？

从熟悉的地方开始着手。这是生活中人们常常采用的方法：希望面对的难题与以前解决过的某个问题是相同的，或者至少类似的；由此就可以获得值得借鉴的经验。面对一个陌生的问题，试图把它和某个熟悉的问题联系起来，借助熟悉的知识和熟悉的方法来解决新问题，是自然的想法。

这种寻找未知问题和已知问题之间联系的思想，可以称为**类比**。更确切的说，“*如果两个系统的各自的各部分之间，存在某种一致的关系的话则称两个系统是可以类比的。*”

如何理解定义中所说的“一致的关系”呢？

如果只简单的把“存在某种一致的关系”理解成一种含糊的相似性。那么类比就完全归结为人的主观的感觉，这种主观上的“似曾相识”是不能够作为分析问题、解决问题的依据的。然而类比的思想的确被广泛的应用于解决各种问题，说明类比的本质是另一种比表面上的相似性更可靠和更有说服力的“一致关系”。事实上，类比是建立在两个问题之间的一种一一对应的映射关系。本文的第一部分，正是试图阐述这一本质上的“一致关系”。

然而两个问题之间的本质联系并不是那么容易得到的。人们在对问题的最初的分析中，注意到的往往还是表面上（甚至只是文字上）的相似性。希望直接得到两个问题之间相互对应和相互转化的关系是不现实的。因此，最初观察到的表面上的相似性虽然有些不可靠，但是至少它能够为分析问

引自参考书目 i（《数学与猜想》）第二章。

题指出方向，由此尝试着建立问题之间的对应关系。正如本文第二部分将要阐述的，利用类比解题的过程是从模糊到清晰，从表面到本质的分析过程。

接下来的两个部分，就将探讨类比过程中，表面的相似和本质的对应之间的关系。

类比的本质，一一对应的关系

类比作为一种分析问题的思想方法，目的是希望将不熟悉的知识转化为熟悉的知识，将未知的问题转化为已知的问题。

如何实现这一转化，取决于如何对两个问题进行类比。如果仅考虑到两个问题表面上的相似性，那么很可能会机械的模仿已知问题的解决方法，来解决新问题。这种想法缺乏严谨可靠的支持，难以保证在实际解题中能够成功。即使成功了，也是知其然而不知其所以然，没有发现问题之间本质的联系，往往得不到最好的解决方法。而当类比过程中两个问题之间存在一种对应关系，未知问题中的所有描述对象和操作，在已知问题中都有与之一一对应的内容，那么整个未知的问题就可以通过这种一一对应的映射关系，转化为已知的问题，也就可以应用已知问题的解决方法来解决它。

下面的例子就是如此。

【“整数拆分”与“因数分解”】

整数拆分：将一个正整数 n，拆分为一组小于 n 的正整数的和（不考虑这组正整数排列的先后次序）。求总共有多少组可能的拆分。

这是一道众所周知的组合计数问题。解决的方法有两种：

1　利用递归的枚举解题模型。(对应程序名 DIVIDE1.PAS)

将问题描述为：求满足等式

$$n = a_1 + a_2 + a_3 + \cdots + a_m \qquad (1 \le a_1 \le a_2 \le \cdots \le a_m < n)$$

的所有正整数组（a_1，a_2，a_3，a_4……，a_m）的总数。为此，可以采用递归枚举的方法逐个确定每一个 a_i 从而求出所有的解，并统计总数。

设 D(k,n)为将 n 拆分成一组不小于 k 的正整数的和的解的数目。

例如，a_1 可以选取 1...[n/2]之间的任意整数，剩下的 n-a_1 可以拆分成不小于 a_1 的若干个整数的和。于是对于 a_1，有 $D(1,n) = \sum_{1 \le a_1 \le [n/2]} D(a_1, n-a_1)$；

而对 a_2，有

$$D(a_1, n-a_1) = \sum_{a_1 \le a_2 \le [(n-a_1)/2]} D(a_2, n-a_1-a_2) \quad \{\text{若拆分成2个以上的数}\}$$
$$+ \ 1 \qquad \{\text{若只拆分成2个数}\}$$

由此不难得出问题的递归求解式

$$D(k,n) = \sum_{k \le i \le [n/2]} D(i, n-i) \ +1;$$

问题的解等价于 $D(1,n)-1$（减一以除去 n = n 的拆分方案）。

问题来自参考书目 ii（《算法设计》）4.2.4。

由于原问题只要给出解的总数，而这一算法在计算过程中求出了所有的解，所以枚举算法的效率在 n 较大的时候是很低的。

2　母函数解题模型。(对应程序名 DIVIDE2.PAS)

设拆分的这组正整数中 1 出现的次数为 x_1，2 出现的次数为 x_2，等等，那么问题可以描述为：求不定方程

$$1x_1+2x_2+\cdots+k*x_k+\cdots+n*x_n=n \qquad (x_i>0)$$

的整数解的总数。于是就可以利用构造母函数，来求不定方程的整数解的个数。方法如下：

拆分中不选取 1 可以表示为 $x^0=1$，取一个 1 表示为 x，取两个 1 为 $x*x=x^2$，取 k 个 1 为 x^k。由加法原理得到选取 1 的所有方案为 $1+x+x^2+x^3+\cdots+x^n$；

不选取 2 可以表示为 1，取一个 2 表示为 x^2，取两个 2 为 $x^2*x^2=x^4$，取 k 个为 x^{2k}，由加法原理得到选取 2 的所有方案为 $1+x^2+x^4+\cdots+x^{2k}+\cdots+x^{2[n/2]}$；

同理，对于正整数 r，选取 k 个表示为 x^{k*r}，由加法原理得到选取 r 的所有方案为　$1+x^r+x^{2r}+\cdots+x^{k*r}+\cdots+x^{[n/r]*r}$；

再由乘法原理，同时选取 1,2,…,n-1 的方案可以表示为

$$\begin{aligned} g(x)=&(1+x+x^2+x^3+\cdots+x^n)\\ &(1+x^2+x^4+\cdots+x^{2k}+\cdots+x^{2[n/2]})\\ &(1+x^3+x^6+\cdots+x^{3k}+\cdots+x^{3[n/3]})\\ &\cdots(1+x^r+x^{2r}+x^{k*r}+\cdots+x^{[n/r]*r})\cdots\\ &(1+x^{n-1}) \end{aligned}$$

由于要使最后的和等于 n，所以 g(x)展开式中 x^n 的系数，就是问题的解。对于这类母函数，无法直接用通项公式计算其展开式的系数，但是可以通过逐次进行的多项式的乘法求解展开式。这在程序实现中相当于一个线性表的归并算法，而计算过程中也不必求出方程实际的解，速度较上一种方法快得多。（程序和运行效率的比较，请见附录）

因数分解：将一个正整数 n，分解为除 1 和它本身的因数的乘积的形式，不考虑因数的先后顺序，求所有可能的分解方案的总数。

将一个正整数分成若干个正整数，这种表述是熟悉的。事实上，这和整数拆分中的表述是相似的。虽然两个问题之间的区别是显然的：一个是拆成乘积的形式；一个是拆成和的形式。但是两个问题都是计算一个整数分成若干个整数的方案数目。

于是，在注意到了两个问题表面上的相似后，确定“因数分解”的问题也可以用枚举法来解决。不妨着手模仿整数拆分的解题模型①，将问题描述为：求满足等式

$$n=a_1*a_2*a_3*\cdots*a_m \qquad (2\le a_1\le a_2\le a_3\le\cdots\le a_m<n)$$

的所有正整数组（a_1，a_2，a_3，a_4……，a_m）的总数。可以发现，两个问题的描述是非常相似的，只是把加号换成了乘号。可以直接做这样的变换而不改动其它内容的关键在于：加法和乘法都满足交换律和结合律。它们是

利用母函数计算不定方程整数解的总数的一般方法，请参见参考书目 iii（《组合数学及其在计算机科学中的应用》）。

两种非常相似的运算。由以上的描述出发，模仿“整数拆分”的 D(k,n)，同样可以定义 F(k,n)为将 n 分解成一组不小于 k 的正整数的积的解的数目，

则容易推导出 $F(k,n)=\sum_{k\le i\le\lfloor\sqrt{n}\rfloor且i\mid n}F(i,n/i)\quad+1$；

而 F(2,n)-1 就是问题的解。

由于采用的是枚举法，在 n 的因子很多时，程序的速度是非常慢的。(对应程序名 P1.PAS)

以上的解题模型和“整数拆分”的方法①是出奇的类似的。然而，机械的模仿所能做到的也只有这些了。它无法解释两个问题为什么如此类似。

然而，有一点已经被提了出来：两者之所以是相似的，关键在于加法和乘法是相似的。那么加法和乘法之间，究竟是怎样的关系呢？

为了回答这个问题，要引入**同构**的定义：

设有两个集合 A，B，“$\otimes$”和“$\oplus$”为分别定义在两个集合上的（二元）运算。那么称 A 和运算“$\otimes$”构成一个代数系统〈A，$\otimes$〉，B 和运算“$\oplus$”构成代数系统〈B，$\oplus$〉。

如果存在一个一一对应的映射 $f(x):A\to B$，使得对于所有的 x∈A，y∈A，都有 $f(x\otimes y)=f(x)\oplus f(y)$，则称代数系统〈A，$\otimes$〉和〈B，$\oplus$〉是同构的。

如果两个代数系统是同构的，那么显然其中一个系统中的问题总可以转化为另一个系统中的问题。

现在借助简单的中学知识，就可以知道〈R，+〉与〈R^+，*〉是同构的：

设 $f(x)=\ln(x)\quad(x>0)$，则有恒有 $f(a*b)=f(a)+f(b)$ 成立。

对数函数 ln(x)就是把乘法转化为加法的工具。

有了这些作为基础，我们就可以很有信心的利用母函数解决因数分解问题。

首先假设整数 n 有除 1 和 n 以外因数共 m 个，为 p_1、p_2、p_3…p_m，而设每个因数 p_i 在分解的式子中出现次数分别为 x_i，那么问题可以描述成：求方程

$$p_1^{x_1}\cdot p_2^{x_2}\cdot p_3^{x_3}\cdot\cdots\cdot p_m^{x_m}=n$$

的非负整数解的数目。接着，利用 ln(x)把乘法转化为加法，对等式两边取对数，就得到对应的线性方程

$$x_1\ln p_1+x_2\ln p_2+x_3\ln p_3+\cdots+x_m\ln p_m=\ln n;$$

欲求它的非负整数解的个数，利用加法原理和乘法原理构造母函数（推导过程参见“整数拆分”的相应段落），

$$\begin{aligned}g(x)=&(1+x^{\ln p_1}+x^{2\ln p_1}+x^{3\ln p_1}+\cdots+x^{[\ln n/\ln p_1]*\ln p_1})\\&(1+x^{\ln p_2}+x^{2\ln p_2}+x^{3\ln p_2}+\cdots+x^{[\ln n/\ln p_2]*\ln p_2})\\&\cdots(1+x^{\ln p_k}+x^{2\ln p_k}+x^{3\ln p_k}+\cdots+x^{[\ln n/\ln p_k]*\ln p_3})\cdots\\&(1+x^{\ln p_m}+x^{2\ln p_m}+x^{3\ln p_m}+\cdots+x^{[\ln n/\ln p_m]*\ln p_m})\end{aligned}$$

则 g(x)展开式的 $x^{\ln n}$ 项的系数就是问题的解。与整数拆分的求解方法②类似的，利用线性表的归并算法，就可以求出解来。(对应程序名 P2.PAS)

与递归枚举算法相比，算法的效率同样也大大的提高了。（程序和运行效率的比较，请见附录）

如果仅仅凭借表面上观察到的相似性，而不清楚乘法和加法代数系统之间存在着同构的关系，是不容易得出利用母函数的解题模型的。

这个例子中最有启发性的，莫过于同构概念的引入。为了说明两个同构的代数系统之间的关系，借助了一个一一映射 $f(x): A \to B$，这个映射使两个系统的不同运算之间产生了一种**对应关系**：

A集合中的 $x \otimes y$ 对应于 B集合中的 $f(x) \oplus f(y)$；

这样 A 集合中关于 $\otimes$运算的每一个问题，都与 B 集合中关于 $\oplus$运算的一个问题相对应。于是可以把〈A，$\otimes$〉中的问题**转化**为〈B，$\oplus$〉中的问题来解决。

实现这种转化正是利用类比思想解题的目的，而寻找这种一一对应的关系则是**类比的实质**。只要发现了这种一一对应的关系，就能够应用解决已知问题的方法去解决新问题。

同构只是代数系统中的概念，然而对应的关系可以应用到其它问题中因此用类比解题是不局限于代数系统之内的。下面的例子就属于另一个不同方面。目的是在进一步说明类比的表象和它的本质之间的关系。

从表面的相似到内在的对应

类比的本质是揭示已知问题与未知问题之间一一对应的关系。但是，正如引言中说的，本质往往不容易被发现，而表面的相似性却首先被观察到。所以，**不应该**忽视表面的相似性；相反，应该沿着相似性所指出的方向开始分析，尝试发现问题之间的本质联系。

在分析下面这个例子的过程中，将着重说明这一点。

【平面分割空间问题】

一个平面把空间分成独立的两个部分，两个平面把空间分成 4 个部分。n 个平面，最多能把整个空间分割成多少个部分。

立体空间的情况是陌生的，而且不容易描述。虽然同样是“分割”问题。但是正常情况下，不会把它和“整数拆分”联系起来，因为两者处理的对象（整数和空间平面）是完全不同的。那么，究竟如何把这个问题和已知的知识联系起来呢？这个时候，即使是表面的相似性，也能够为解题提供一条线索。

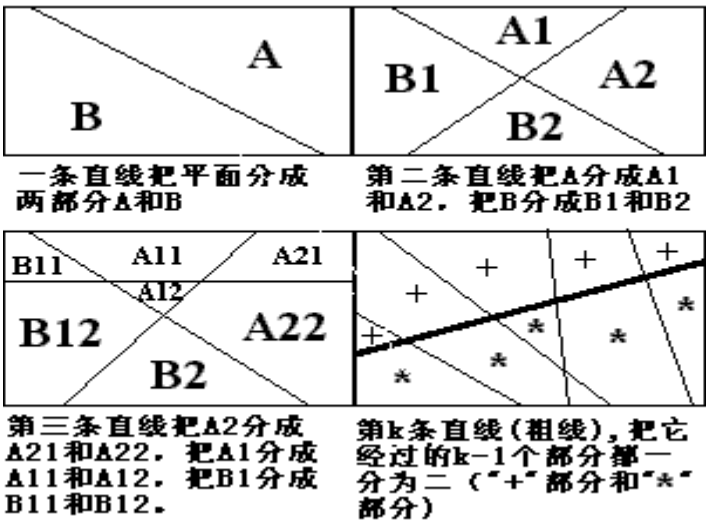
类比的目的是将未知的、复杂的东西化为已知的、简单的东西。空间上的问题是复杂的，而平面上的问题则是简单的。而且空间和平面的关系，与平面和直线的关系，是相似的。平面把空间分割成两个部分，直线也把平面分割成两个部分。

于是得到了另一个与例题相类似的问题：n 条直线，最多能把整个平面分割成多少个部分。

这道题容易解决得多。它的解决方法是利用数学归纳法（如图）：

问题来自参考书目 i（《数学与猜想》）第三章。

这实际上是本届（第五届）分区联赛初赛的一道填空题。



一条直线，把平面分割为两个部分（A 和 B）；

增加一条直线，它与第一条直线相交，被分为两段射线，每一段射线又把所在的区域（例如 A）分成两部分（A1 和 A2）；因此增加了 2 个部分；共有 2+2=4 个部分。

再增加一条直线，它与前两条直线相交，被分为三段，每一段又把所在区域（例如 B1）分成两部分(B11 和 B12)；总共增加了 3 个部分；共有 4+3=7 个部分；

由此类推，设前 k-1 条直线共把平面分为 A_{k-1} 个部分。加入第 k 条直线，与前 k-1 条直线相交，被分为 k 段，每一段把所在区域分为两部分；总共增加了 k 部分；总共有 $A_{k-1}+k$ 个部分。

于是得到了递推关系 $A_1=2$； $A_k=A_{k-1}+k$；

由此不难计算出通项公式：

$A_k=1+\frac{(1+k)k}{2}$；

于是直线分割平面的问题就解决了。

解决这一问题对“平面分割空间问题”有何帮助？既然空间和平面，与平面和直线的关系相似，那么直接把平面换成空面，把直线换成平面，就可以直接用以上的过程来证明未知的问题了吗？显然是不可以的。这仅仅是根据主观的相似得出的机械的模仿，是错误的。

但是如果仔细分析一下错误的原因，将会发现问题的关键：线线相交得到的是**交点**，面面相交得到的是**直线**，k 个点把直线分成 k+1 个部分，而 k 条直线则不是简单的把平面分成 k+1 个部分。

事实上，k 条直线最多把平面分成 A_k 个部分！

因此两个问题真正的相似之处在于，一个为了计算直线把平面分割成几部分，先计算这条直线被点（线线交点）分割成几部分，把二维的问题化为一维的问题；另一个要计算平面把空间分割成几部分，先计算这个平面被线（面面交线）分割成几部分，把三维的问题化为二维的问题。而二维的问题是已经被解决了的；于是反过来，三维的问题也解决了。

同样是利用数学归纳法：

一个平面把空间分割成两部分；

设前 k-1 个平面共把空间分割成 B_{k-1} 个部分。加入第 k 个平面后，与前 k-1 个平面相交，共有 k-1 条交线。k-1 条交线把这个平面被分为几个块呢？这实际上就是刚刚解决过的问题：k-1 条直线，把平面分割成 $A_{k-1}=1+\frac{k(k-1)}{2}$ 部分。这 $1+\frac{k(k-1)}{2}$ 块平面，分别把所在原来空间一分为二，

总共增加了$1+\frac{k(k-1)}{2}$个部分；于是，k 个平面总共把空间分割成$B_{k-1}+1+\frac{k(k-1)}{2}$个部分。

于是得到了递推关系 $B_1=2$；$B_k=B_{k-1}+1+\frac{k(k-1)}{2}$；

利用这一递推关系来编写程序，不难求出 B_n，而且即便对很大的 n，程序的运算速度仍然是很快的。

（当然，也可以计算出 B_n 的通项公式，这实际上是一个二阶等差数列的和：

$$B_n=1+n+\frac{n(n-1)}{2}+\frac{n(n-1)(n-2)}{6}$$）

在这个例子中的一一对应的关系，虽然不如同构的概念中那么容易地用式子表达出来，但还是很清楚的：n 个平面中的任意一个平面，截其它 n-1 个平面得到的所有 n-1 条交线，把这个平面分成 A_{n-1} 部分，而这正**对应**于是第 n 个平面分割空间得到的**增量**。或者说，

$\Delta B_n=B_n-B_{n-1}=A_{n-1}$；

在这个例子中，最初找不到什么熟悉的类似问题，但是从简单和便于分析的角度着手，构造了一个平面上的与之相似的问题。此时，相似性可以说是“立体”和“平面”的类比，这是非常模糊的。因为几乎没有确切的理由，能表明这样的类比可以使问题相互转化；这种相似性主要依据的是以往解决问题的经验（化空间问题为平面问题，在中学的立体几何中是经常用到的手法），另一个原因则是为了简化问题。

接着，在解决平面的问题之后，面对的困难是：如何把解决平面上问题的经验应用到空间的问题中。如果仅仅根据表面的相似性，机械的模仿将导出一种错误的方法。然而这种尝试却不是没有价值的。通过比较正确的论证和错误的论证之间的差别，原来的简单而模糊的类比得到了完善和澄清：把“三维降低到二维的过程”和“二维降低到一维的过程”做类比。由此，两个问题之间的联系变得很清楚，于是得出了完整的推算过程。

事实上，这道例题的解决，和近年来不少三维的计数问题的思考方法是非常类似的：先考虑把二维化为一维的情况；再用相同的方法（类比），把三维的问题化为二维的问题来解决。这种策略被形象的称为“**降维**”的策略。

以上“因数分解”和“平面分割空间问题”的例子，都是借助类比解决的。而且在最初，都把探究问题之间的某种相似性作为分析问题的**线索**，这样做的目的是希望能够把未知的问题和熟悉的知识联系起来，以提供我们解决问题的可能性。然而正如在这两个例子中见到的，这种从问题的表面上得出的相似性，往往不是本质的，并且可能是有缺陷的。所以，这种表面现象上的类比只是为解题指出了可行的方向。为了最终能够把未知的问题转化为已知的问题，需要揭示两个问题之间更深层的类比，即建立从一个问题到另一个问题的对应关系。

在利用类比解题的整个过程中，表面的相似和内在的对应是紧密的

CTSC’99 的论文《隐蔽化、多维化、开放化》（石润婷）中做了十分详细的分析。

联系在一起的。相似性在浅层，对应关系在深层。对应关系是解决问题的关键，而发现对应关系的途径是从表面的相似性出发的。

总结

信息学竞赛涉及到大量的不同算法，对于每一种算法的特点和应用技巧，老师和前辈们都已经进行了大量的分析和研究，并且总结出了许多完整的理论。那么，在分析问题和解决问题的思路方面，是否也有一定的规律和方法呢？出于这样的考虑，本文没有就某种具体的算法进行分析，而是希望能够在分析问题的**思想方法**上有所探讨。在这方面，许多数学上的技巧（例如构造、化归、归纳法）都是值得借鉴的，本文所主要分析的类比思想，也是其中之一。

本文的第一部分，通过利用类比解题的例子，首先指出仅仅表面的相似性并不能真正体现两个问题之间的实质性联系；然后，从代数系统的同构的定义中提取出类比的本质，即两个系统的各个部分之间一一对应的关系。这种一一对应的关系，使一个系统中的一个操作（运算）对应于另一个系统中的另一个操作（运算）。由此，可以把关于一个陌生的系统中的新问题，转化为熟悉的问题来解决。

一一对应的关系是类比本质的一面，而主观上的相似性则是类比的表象。同时，正如在本文的第二部分所阐述的，这两者不是完全对立，而是互相联系的。没有主观的相似性的引导，类比的本质很难被发现。整个类比的过程，从含糊（主观上的）的相似到清晰的（概念上的）对应关系，把已知的问题与未知问题逐渐联系起来，最后使两个问题之间能够互相转化，从而把已知问题的解题模型应用到未知问题的解决中去。

利用类比解题并不是某种固定的算法，而是分析问题时的一种思考方法，因此没有什么定式，但也有规律可循。类比的目的，是希望化复杂的问题为简单的问题，化陌生的问题为熟悉的问题。为了实现这一转化，最初，可能是按照某种相似性，把两个不同的问题联系起来，并尝试着模仿解决已知问题的方法，来解决新问题。在这一模仿的过程中，如果能够明晰两个问题之间存在的对应关系，就能够实现两个问题之间的转化；否则，如果仅仅是机械的模仿，往往是难以解决问题的。

类比，是一种把已知的种种信息和知识，与陌生的问题联系起来的思路。这既依靠选手解题时发散思维的能力，又取决于选手在平时的学习和训练中所接触知识的深度和广度；而类比解题的基础也要求选手拥有与未知问题相类似的知识，这就需要选手在平时的学习和训练中既有扎实的基本功，又有广泛的知识面。

因此，类比这种解题思路，对选手的要求正是“**能力**”+“**基本功**”。

参考书目

i. **《数学与猜想》（《Mathematics and plausible**

reasoning》)，G·Polya 著，李心灿、王日爽、李志尧译，科学出版社，1984。

ii. **《算法设计》，蒋新儿著，全国青少年计算机课外活动辅导员培训教材编委会，1996。**

iii. **《组合数学及其在计算机科学中的应用》，庄心谷，西安电子科技大学出版社，1989。**

附录

“整数拆分”问题利用不同算法的程序的效率比较：

N	问题的解	递归枚举算法 (divide1.pas) 的耗时(秒)	利用母函数的多项式乘法算法(divide2.pas)的耗时(秒)
5	7	0.00	0.00
10	42	0.00	0.00
50	204226	0.05	0.00
80	15796476	7.14	0.00
90	56634173	25	0.00
95	104651419	47	0.00
100	190569292	120	0.05

“因数分解”问题不同算法的效率比较：

N	问题的解	递归枚举算法 (p1.pas) 的耗时（秒）	利用母函数的多项式乘法(p2.pas)算法的耗时(秒)
12	3	0.00	0.00
24	6	0.00	0.00
1048576	626	0.05	0.05
9261000	27035	0.22	0.49
1073741824	5603	0.33	0.11
1803601800	558189	10.05	3.19
479001600	2787809	16.65	2.25
518918400	2756006	18.19	3.68
1556755200	6348742	45.12	4.84

程序清单：

[DIVIDE1.PAS]:

```
PROGRAM Divide_1;                    { 整数拆分的递归枚举算法的程序 }
VAR     n      :INTEGER;
```

```
        total :LONGINT;          { 解的总数 }

PROCEDURE init;                  { 初始化，读入 n }
BEGIN
  readln(n);
  total:=0;
END;

PROCEDURE D(k,l:INTEGER);        { 递归过程 }
VAR i:INTEGER;
BEGIN
  FOR i:=k TO l div 2 DO
      D(i,l-i);                  { 把 l-i 拆分为不小于 i 的若干个整数的和 }
  Inc(total);                    { 若 l 不再拆分为更小的整数，则得到一个解 }
END;

{ 主程序 }
BEGIN
  init;                          { 初始化 }
  D(1,n);                        { 把 n 拆分为不小于 1 的若干整数的和 }
  writeln(total-1);              { 解的总数减一，除去 n=n 的拆分情况 }
END.
```

[DIVIDE2.PAS]:

```
PROGRAM Divide_2;                { 整数拆分的母函数算法的程序 }
VAR     n     :INTEGER;
        total :LONGINT;          { 解的总数 }

PROCEDURE init;                  { 初始化，读入 n }
BEGIN
  readln(n);
  total:=0;
END;

PROCEDURE work;                  { 主过程 }
TYPE    TForm = array[0..1000] of LONGINT;
                                 { 多项式类型，数组下标对应于幂指数 }
VAR     X1,X2 : TForm;
        i,j,k : INTEGER;
BEGIN
  fillchar(X1,sizeof(X1),0);
  FOR i:=0 TO n DO X1[i]:=1;     { X1 = 1+x+x^2+x^3+...+x^n  }
  FOR i:=2 TO n-1 DO
        { 以下计算 X1*（1+x^i+x^2i+x^3i+...+x^[n/i]）暂时存入 X2，再保存到 X1 }
  BEGIN
       X2:=X1;                   { X2 = 1*X1 }
       j:=i;
       WHILE (j<=n) DO
       BEGIN
                { 以下计算 X2= X2 + x^j * X1 }
            FOR k:=0 TO n DO X2[k+j]:=X2[k+j]+X1[k];
```

```
            j:=j+i;
        END;
        X1:=X2;
  END;
  total:=X1[n];                    { x^n 的系数就是问题的解的总数 }
END;

{ 主程序 }
BEGIN
  init;                            { 初始化 }
  work;                            { 求解 }
  writeln(total);                  { 输出 }
END.
```

[P1.PAS]:

```
PROGRAM  p1;                       { 因数分解的递归枚举算法的程序 }
VAR  n     : LONGINT;
     total : LONGINT;              { 解的总数 }

PROCEDURE init;                    { 初始化，读入 n }
BEGIN
  readln(n);
END;

PROCEDURE P(k,l:LONGINT);          { 递归过程 }
VAR i:LONGINT;
BEGIN
  FOR i:=k TO Trunc(Sqrt(l)) DO
      IF  l mod i=0  THEN          { 如果 i 是 l 的约数 }
          P(i,l div i);            { 把 l/i 分解成不小于 i 的若干个整数的积 }
  inc(total);                      { 若 l 不再分解成更小的整数，则得到一个解 }
END;

{ 主过程 }
BEGIN
  init;                            { 初始化 }
  P(2,n);                          { 把 n 分解为不小于 2 的若干个整数的积 }
  writeln(total-1);                { 输出解的总数减一，除去 n=n 的情况 }
END.
```

[P2.PAS]:

```
PROGRAM  p2;                       { 因数分解的母函数算法的程序 }
VAR   n     : LONGINT;
      total : LONGINT;             { 解的总数 }

PROCEDURE init;                    { 初始化，读入 n }
BEGIN
  readln(n);
END;
```

```
PROCEDURE work;                    { 主过程 }
TYPE TForm = array[1..1500] of RECORD        { 多项式类型 }
                                 s,exp:LONGINT;
                                 { s 为系数，exp 为幂指数上 ln(x)的自变量 x }
                               END;
VAR  i               : LONGINT;
     F1,F2,F3        : TForm;
     top1,top2,top3 : INTEGER;   { 对应于多项式 F1，F2，F3 的长度 }

    PROCEDURE AddForm(p:LONGINT);
                        { 计算 F1 = F1 * （1+x^ln(p)+x^2ln(p)+x3^ln(p)+... ）}
                        { 对于多项式 F1 所有项 x^ln(p)，只保存满足 p 是 n 约数的项 }
    VAR j,k,l,m:LONGINT;
    BEGIN
      j:=1;     { j = p^i  }
      m:=n;
      WHILE m mod p=0 DO          { 当 m 含有因子 p，j*p 是 n 的约数的时候 }
      BEGIN
           m:=m div p;                  j:=j*p;
           { 以下计算多项式 F1*x^ln(j)，并把它与 F2 归并（暂存在 F3 中） }
           l:=1;
           top3:=0;
           fillchar(F3,sizeof(F3),0);
           FOR k:=1 TO top1 DO
           IF m mod F1[k].exp=0 THEN
           { m 含有因子 F1[k].exp，将 F1 中的第 k 项与 x^ln(j)的乘积归并入 F3 }
           BEGIN
                WHILE (l<=top2) and (F2[l].exp<F1[k].exp*j) DO
                    { 将 F2 幂指数小于 ln(j*F1[k].exp)的项先归并入 F3 }
                BEGIN
                     inc(top3);         F3[top3]:=F2[l];
                     inc(l);
                END;
                inc(top3);
                IF F2[l].exp=F1[k].exp*j THEN { 合并同类项 }
                BEGIN
                     F3[top3].exp:=F2[l].exp;
                     F3[top3].s:=F3[top3].s+F1[k].s;
                     inc(l);
                END ELSE BEGIN
                              F3[top3].s:=F1[k].s;
                              F3[top3].exp:=F1[k].exp*j;
                         END;
           END;
           WHILE (l<=top2) DO             { 将 F2 中没有归并的部分并入 F3 中 }
           BEGIN
                inc(top3);         F3[top3]:=F2[l];
                inc(l);
           END;
           F2:=F3;               top2:=top3;
                { 将归并的结果保存到 F2，即 F2=F2 + F1*x^ln(j) }
```

```
      END;
      F1:=F2;                top1:=top2;
                { 将多项式乘法的最后的结果保存到F1 }
    END;

BEGIN
  fillchar(F1,sizeof(F1),0);
  top1:=1;
  F1[1].s:=1;                    F1[1].exp:=1;  { F1 = 1 }
  top2:=top1;                    F2:=F1;        { F2 = F1 = 1 }
  FOR i:=2 TO trunc(sqrt(n)) DO
      IF n mod i=0 THEN          { 如果i是n的约数 }
      BEGIN
           AddForm(i);           { F1=F1*(1+x^ln(i)+x^2ln(i)+x^3ln(i)+...+x[ln(n)/ln(i)]) }
           IF i*i<>n THEN        { 如果i不是n的平方根，n/i也是n的约数 }
              AddForm(n div i); { F1 = F1*(1+x^ln(n/i)+x^2ln(n/i)+... ) }
      END;
  i:=top1;
  WHILE (i>0) and (F1[i].exp<>n) DO dec(i);
                                 { i的ln(n)次幂的系数就是解的总数 }
  IF i=0 THEN total:=0 else total:=F1[i].s;
END;

{ 主程序 }
BEGIN
  init;                          { 初始化 }
  work;                          { 计算 }
  writeln(total);                { 输出 }
END.
```