算法竞赛

进阶指南

李煜东 著

中原出版传媒集团
中原传媒股份公司
河南电子音像出版社

算法竞赛

进阶指南

李煜东 著

中原出版传媒集团
中原传媒股份公司
河南电子音像出版社

图书在版编目（CIP）数据

算法竞赛进阶指南 / 李煜东著 . —郑州：河南电子音像出版社，2017.10

ISBN 978-7-83009-313-6

Ⅰ . ①算… Ⅱ . ①李… Ⅲ . ①计算机算法－指南 Ⅳ . ①TP301.6-62

中国版本图书馆 CIP 数据核字（2017）第 242365 号

算法竞赛进阶指南

出版发行：河南电子音像出版社
地　　址：郑州市郑东新区祥盛街 27 号
邮政编码：450016
电　　话：0371-65788820, 65788822, 65788829
责任编辑：米军阳
责任校对：孙春芳
印　　刷：河南省诚和印制有限公司
开　　本：787 毫米 ×1092 毫米　1/16
印　　张：31.125
字　　数：435 千字
版　　次：2018 年 1 月第 1 版
印　　次：2019 年 4 月第 4 次印刷
印　　数：10000-13000 册
定　　价：79.00 元
书　　号：ISBN 978-7-83009-313-6

0xFF 前言——如何阅读本书

探索一门学问有三个层次：求其解，知其原因，究其思维之本。通俗地讲，就是“怎么做”“为什么是对的”“怎样才能想到这么去做”。在计算机科学中，前两者分别对应算法的步骤和证明。而长久以来，后者时常与“天赋”一词相关联。本书在全面讲解算法与数据结构知识点的同时，致力于模型构建与思路分析，帮助读者厘清思维过程的线索和脉络，建立一棵“技能树”，从容地面对算法竞赛的各项挑战，为未来更高层次的程序设计与研究奠定坚实的基础。

撰写背景

我当年准备参加信息学竞赛时，所在地区的教学资源相对不足，在大约一年半的时间里，依靠 3000 道以上的巨大刷题量，才从一名连深搜都写不对的初学者，成长为 NOI 金牌得主，并入选国家集训队。2012 年以来的每个寒、暑假，我都参与了相关竞赛的授课、命题等工作。为了把知识清晰、透彻地讲解给学生，我不得不转变自己的角色，从“会做题”的选手，变成一位“知其所以然”的教师。我发现在高中阶段的竞赛学习中，因自己死记硬背、不求甚解，导致对许多知识的理解都存在偏差。每次重新审视一个解法、更新一次课件，我都对“分析证明”的重要性和“怎样想到一种解法”的思路历程有了更加深刻的认识。

在思维的迷宫里，有的人凭天生的灵感直奔终点；有的人以持久的勤勉，铸造出适合自己的罗盘；有的人迷失了方向，宣告失败。作为算法竞赛的过来人，我很欣喜地看到大多数选手无论结果如何，都保持着对程序设计的热情和喜爱，在日后的学习、工作中继续选择了信息科学方向。诚然，悟“道”还需个人，但准备算法竞赛的时间毕竟非常有限。我想在彻底从算法竞赛界退役之前，有必要尽一点儿微薄之力，把自己数年来的所寻所得汇集成卷，只求读者能从中受益些许。我深知作为一名指导者，想要代替时间与见识的积淀，给学习者传播“怎样才能想到这么去做”，是一件难上加难的事情。因此，希望读者始终保持独立的思考习惯和坚定的批判精神，对于每种模型，不仅能用数学语言推导其正确性，更能自然而然地“说服”自己，使整个程序设计过程完全融于自己的思维体系之中。

前期准备

《算法竞赛进阶指南》，顾名思义，是一本侧重进阶类的书籍。认真阅读本书的每

一个知识点，动手完成每一道例题和习题，并加以归纳、理解，读者将会从一名算法竞赛的入门级选手，达到 NOIP 一等奖获得者的中、高级水平，并具有相当的自我学习与思考能力。在此基础上，读者只需涉猎一些新的领域，扩大知识范围，辅以实战训练，不难继续提升到 NOI 省队平均水平，或具有组队夺得 ACM-ICPC 区域赛金牌的能力。

在阅读本书前，读者应该先对算法竞赛的形式有所了解，掌握至少一门编程语言，能够运用朴素算法求解最基本的问题。最好是大致浏览过一本入门类教材，有 100 道题左右的练习经验，或者三个月到半年的课余学习经历。读者可以对照以下清单进行自我检验：

- ✓ 熟悉 C++语言的基本语法结构，能够使用 C++语言编写程序。
- ✓ 会使用朴素算法，如枚举、模拟等方法。
- ✓ 理解“递推”的概念，能循环计算简单的递推式。
- ✓ 掌握几种常见的排序算法，如选择排序、冒泡排序、快速排序等。
- ✓ 熟练使用数组，知道链表、栈、队列的基本结构与用法。
- ✓ 能够借助一些 C/C++内置函数，进行简单的字符串处理。
- ✓ 了解树、图结构，以及其中的常见术语与专有名词。
- ✓ 接触过动态规划，会做数字三角形、LIS、LCS 等入门题。
- ✓ 具有不低于高中一年级水平的基本数学素养。

即使在上述条件中有所欠缺，也不代表一定不能读懂本书。读者可以在遇到不熟悉的内容时，随时使用 Google、Bing（国际版）等搜索引擎查阅资料，辅助学习。

阅读建议

本书由“基本算法”“基本数据结构”“搜索”“数学知识”“数据结构进阶”“动态规划”“图论”“综合技巧与实践”八章组成，每章对应着算法与数据结构中一个较为独立的领域，并进一步划分为若干节，每节都是所在领域知识体系中较为完整的一块，各章节以十六进制整数编码。整本书尽量以循序渐进的方式编写，对于不得不涉及的后续章节的内容均有所标注，读者可以从前到后依次学习。

除此之外，读者也可以按照知识点从易到难的顺序，先阅读第 0x00、0x10 和 0x70 章，再阅读第 0x21、0x22、0x25、0x31~0x33、0x36、0x41、0x51~0x53、0x61、0x62 节，接下来阅读第 0x20 章的剩余部分和第 0x42~0x46 节，最后阅读第 0x30、0x40、0x50 和 0x60 章的剩余部分。

1. 知识点讲解与例题

每节内容的主干由“知识点讲解部分”以及与之紧密相关的例题组成，读者应当配套进行学习。不过，还是鼓励读者先花费适当的时间独立思考，再查看例题的详细解

答。算法竞赛最终要落实在程序设计上，请读者务必动手完成每一道例题的代码实现，方能达到期望的训练效果。

2. 知识点清单与拓展

每章的最后一节都包含一份详细的知识点清单，供读者在复习时对照、检验。这些清单基本涵盖了信息学竞赛省级联赛的知识范围，尤其在动态规划、图论方面已经具有了相当的深度。在确保掌握了清单中列举的所有内容后，读者应该自主完成章末的练习题。如果遇到瓶颈，可以谨慎利用搜索引擎查阅题解。

认真阅读完全书后，若读者仍想进一步提高，可以重点关注高级数据结构、网络流、计算几何这三个本书涉及不多的领域，并进一步提升数学水平。

3. 光盘

大部分例题和习题都标注了来源，可以直接在 Online Judge 上提交。对于部分未公开或者所属 OJ 不稳定的题目，本书配套的光盘提供了测试数据。光盘中也包含多数题目的原版题面和参考程序。0x7F 节总结并列举了全书涉及的所有例题、习题，包括题目版权方、在线提交地址、光盘是否提供测试数据等。

4. 关于英文

许多题目的原始题目描述为英文。对于例题，本书提供了题目大意的翻译；对于习题，仅在光盘中提供原版题面。随着在计算机科学与技术领域学习的不断深入，读者查阅文献、进行学术交流时，遇到英文的概率将会越来越大。希望读者借此机会，提前感受英文环境，掌握一些算法与数据结构方面的英文术语，切勿对英文题目抱有逃避心态。

5. 更新与反馈

GitHub 的 Tedukuri 项目作为《算法竞赛进阶指南》社区，将随时发布有关本书的任何最新消息、内容更正等，同时也欢迎读者参与共建。读者可以通过在 GitHub 上建立 Issue 的方式，指出本书存在的问题。读者也可以 Fork 本项目，上传自己的修改，然后通过提交 Pull Request 的方式，请求合并到原项目中。

本书在 Contest Hunter 上有专属题库，将不定期上传书中的题目。

本书还有读者 QQ 群 650836280。读者可以与其他学习者进行即时交流，更快地找到问题的答案，了解更广泛的思路，追逐竞赛前沿热点。

关于在线资源的详细访问方式，请参阅 0x7F 节“附录”。

在阅读过程中，希望读者时刻把自己视作一位科学研究者，切忌自欺欺人、口头浮夸、懒于思考、直接复制；也希望读者不仅关注结论、模型、算法本身，更要关注它们的分析证明与内在逻辑，建立自然、完整且不容置疑的思维体系。

致谢

本书自 2016 年 12 月动笔，2017 年 10 月完稿。全书约 40 万字。初稿总编辑时长达 60000 分钟，编写期间得到了许多老师、同学和朋友的支持与帮助。

本书第 0x00 章“基本算法”、第 0x10 章“基本数据结构”、第 0x30 章“数学知识”由北京大学石昊悦同学审阅，第 0x20 章“搜索”、第 0x40 章“数据结构进阶”、第 0x60 章“图论”、第 0x70 章“综合技巧与实践”由哈尔滨工业大学苏佳轩同学审阅，第 0x50 章“动态规划”由石家庄市第二中学李晶老师审阅。三位审稿者认真、负责，以严谨的科学态度尽力保证本书所讲内容的正确性，读者所看到的每一个文字都饱含着他们付出的努力。

在编写和审阅期间，北京大学李源昊同学，石家庄市第二中学郭资政、杨卓毅、杨思祺同学以及复旦大学刘炎明同学提前阅读了本书部分内容，多次提出宝贵的修改意见。另外，本书的多道例题、习题出自北京旷视科技有限公司杜宇飞先生、清华大学毕克同学之手。他们是本书的直接贡献者。

当然，本书的成功出版离不开河南电子音像出版社编辑部米军阳主任以及出版社多名编辑的辛劳，他们自始至终推动着本书的编写进展。大视野在线测评为本书开辟讨论专帖，为本书的后续服务做了许多工作。

最后，我想感谢我的父母，感谢我信息学竞赛时期的教练潘雪峰老师、ACM-ICPC 竞赛时期的教练罗国杰老师，没有他们的教导，就不可能有今日本书的面世。

算法竞赛博大精深，而作者时间、能力有限，只做了一点儿微小的工作，书中难免有遗漏或不足之处，还望读者海涵并不吝赐教。期待本书再版之时，能够更臻完善。

作　者　李煜东

2017 年 10 月 8 日于北京

目　录

0x00 基本算法

在本章中，我们将会学习位运算、递推、递归、二分、排序、倍增、贪心等算法，以及前缀和、差分、离散化等技巧。它们简洁易懂，却又蕴含着巧妙而深刻的思想，作为构建算法与数据结构体系的基本单位，贯穿于整个程序设计竞赛的始末。永远不要小瞧这些基本思想，因为认真学习它们对于理解更加高深的算法有着极大的帮助，甚至还有可能帮你开启发现并创造新算法的门窗。

0x01 位 运 算

bit 是度量信息的单位，包含 0 和 1 两种状态。计算机的各种运算最后无不归结为一个个 bit 的变化。熟练掌握并利用位运算，能够帮助我们理解程序运行中的种种表现，提高程序运行的时空效率，降低编程复杂度。

读者可能已经发现，本书的章节目录是以 0x00~0xFF 这些由数字 0~9 与字母 A~F 表示的 2 位十六进制整数进行编号的，其中“0x”表示十六进制。正文即本章由 0x00 开始，前言分配序号 0xFF，后记分配序号 0x7F。这就是以最高二进制位为正负符号位的“补码”形式表示的 8 位二进制数。在 C++中，8 位二进制数对应 char 类型，范围为−128~127，其中 0xFF 代表−1，0x7F 代表最大值 127。

在阅读本节之前，读者应该对以下算术位运算有一个初步的认识[①]：

与	或	非	异或
and, &	or, |	not, ~	xor

它们不局限于逻辑运算，均可作用于二进制整数。为了避免混淆，本书统一用单词 xor **表示异或运算**，而用符号“^”**表示乘方运算**（虽然该符号在 C++中表示异或）。

另外，在 m 位二进制数中，为方便起见，通常称**最低位为第 0 位，从右到左依此类推，最高位为第 $m-1$ 位**。本书默认使用这种表示方法来指明二进制数以及整数在二进制表示下的位数。

① 有关“与”“或”“非”“异或”等算术位运算的基础知识，可参见配套光盘提供的参考资料。

下面我们以 32 位二进制数，即 C++的 int 与 unsigned int[①] 类型为例详细介绍计算机中的整数存储与运算。

◆ 补码

32 位无符号整数 unsigned int：

直接把这 32 位编码 C 看作 32 位二进制数 N。

32 位有符号整数 int：

以最高位为符号位，0 表示非负数，1 表示负数。

对于最高位为 0 的每种编码 C，直接看作 32 位二进制数 S。

同时，定义该编码按位取反后得到的编码 $\sim C$ 表示的数值为 $-1-S$。

32 位补码表示	unsigned int	int
000000…000000	0	0
011111…111111	2 147 483 647	2 147 483 647
100000…000000	2 147 483 648	−2 147 483 648
111111…111111	4 294 967 295	−1

可以发现在补码下每个数值都有唯一的表示方式，并且任意两个数值做加减法运算，都等价于在 32 位补码下做**最高位不进位**的二进制加减法运算。发生算术溢出时，32 位无符号整数相当于**自动对 2^{32} 取模**。这也解释了“有符号整数”算术上溢时出现负数的现象。

补码也被称为“二补数”。还有一种编码称为反码，也叫“一补数”，直接把 C 的每一位取反表示负 C。补码与反码在负数表示中，绝对值相差 1。例如，在上表中，第 1、4 行是一对反码，第 2、3 行是一对反码。作为整数编码、存储和运算的方式，补码与反码相比有许多优势。除了上面提到的“自然溢出取模”之外，补码重点解决了 0 的编码唯一性问题，能比反码多表示一个数，同时减少特殊判断，在电路设计中极其简单、高效。

形式	加数	加数	和
32 位补码	111…111	000…001	~~(1)~~000…000
int	−1	1	0
unsigned int	4 294 967 295	1	0 (mod 2^{32})

① 在信息学竞赛中，我们一般认为 int 即为 32 位有符号整数，unsigned int 即为 32 位无符号整数。在大多数现代计算机上也确实如此。不过，在 C++标准中，仅规定了 int 和 unsigned int 的位数不小于 16。请读者注意，直接把 int 和 unsigned int 与 32 位整数画等号实际上是不严谨的。

形式	加数	加数	和
32 位补码	011…111	000…001	100…000
unsigned int	2 147 483 647	1	2 147 483 648
int	2 147 483 647	1	−2 147 483 648

因为用二进制表示一个 int 需要写出 32 位，比较繁琐，而用十进制表示，又不容易明显地体现出补码的每一位，所以在程序设计中，常用十六进制来表示一个常数，这样只需要书写 8 个字符，每个字符（0~9，A~F）代表补码下的 4 个二进制位。C++的十六进制常数以“0x”开头，“0x”本身只是声明了进制，“0x”后面的部分对应具体的十六进制数值。例如：

32 位补码	int（十进制）	int（十六进制）
000000…000000	0	0x0
011111…111111	2 147 483 647	0x7F FF FF FF
00111111 重复 4 次	1 061 109 567	0x3F 3F 3F 3F
111111…111111	−1	0xFF FF FF FF

上表中的 0x3F 3F 3F 3F 是一个很有用的数值，它是满足以下两个条件的最大整数。

1. 整数的两倍不超过 0x7F FF FF FF，即 int 能表示的最大正整数。
2. 整数的每 8 位（每个字节）都是相同的。

我们在程序设计中经常需要使用 `memset(a, val, sizeof(a))` 初始化一个 int 数组 a，该语句把数值 val（0x00~0xFF）填充到数组 a 的每个字节上，而 1 个 int 占用 4 个字节，所以用 memset 只能赋值出“每 8 位都相同”的 int。

综上所述，0x7F 7F 7F 7F 是用 memset 语句能初始化出的最大数值。不过，当需要把一个数组中的数值初始化成正无穷时，为了避免加法算术上溢或者繁琐的判断，我们经常用 `memset(a, 0x3f, sizeof(a))` 给数组赋 0x3F 3F 3F 3F 的值来代替。该语句在后续章节的参考程序中将会多次出现。

◆ 移位运算

左移

在二进制表示下把数字同时向左移动，低位以 0 填充，高位越界后舍弃。

$$1 \ll n = 2^n, \qquad n \ll 1 = 2n$$

算术右移

在二进制补码表示下把数字同时向右移动，高位以符号位填充，低位越界后舍弃。

$$n \gg 1 = \left\lfloor \frac{n}{2.0} \right\rfloor$$

算术右移等于除以 2 向下取整，$(-3) \gg 1 = -2$，$3 \gg 1 = 1$。

值得一提的是，“整数/2”在 C++中实现为“除以 2 向零取整”，$(-3)/2 = -1$，$3/2 = 1$。请读者自己尝试使用 C++编译器编译运行类似的语句，检查运算结果。

逻辑右移

在二进制补码表示下把数字同时向右移动，高位以 0 填充，低位越界后舍弃。

C++语法没有规定右移的实现方式，使用算术右移还是逻辑右移由编译器决定。一般的编译器（较新版本的 GNU C++与 Visual Studio C++）均使用算术右移。除非特殊提示，我们默认右移操作采用算术右移方式实现。

【例题】 a^b CH0101

求 a 的 b 次方对 p 取模的值，其中 $1 \le a, b, p \le 10^9$。

相关题目：POJ1995 Raising Modulo Numbers

根据数学常识，每一个正整数可以唯一表示为若干指数不重复的 2 的次幂的和。也就是说，如果 b 在二进制表示下有 k 位，其中第 $i(0 \le i < k)$ 位的数字是 c_i，那么：

$$b = c_{k-1}2^{k-1} + c_{k-2}2^{k-2} + \cdots + c_0 2^0$$

于是：

$$a^b = a^{c_{k-1}*2^{k-1}} * a^{c_{k-2}*2^{k-2}} * \cdots * a^{c_0*2^0}$$

因为 $k = \lceil \log_2(b+1) \rceil$（其中$\lceil\ \rceil$表示上取整），所以上式乘积项的数量不多于 $\lceil \log_2(b+1) \rceil$ 个。又因为：

$$a^{2^i} = \left(a^{2^{i-1}}\right)^2$$

所以我们很容易通过 k 次递推求出每个乘积项，当 $c_i = 1$ 时，把该乘积项累积到答案中。b & 1 运算可以取出 b 在二进制表示下的最低位，而 b >> 1 运算可以舍去最低位，在递推的过程中将二者结合，就可以遍历 b 在二进制表示下的所有数位 c_i。整个算法的时间复杂度[①]为 $O(\log_2 b)$。

```
int power(int a, int b, int p) { // calculate (a ^ b) mod p
    int ans = 1 % p;
    for (; b; b >>= 1) {
        if (b & 1) ans = (long long)ans * a % p;
```

① 关于“时间复杂度”的定义与相关知识，可参见配套光盘提供的参考资料。

```
        a = (long long)a * a % p;
    }
    return ans;
}
```

在上面的代码片段中，我们通过“右移（>>）”“与（&）”运算的结合，遍历了 b 的二进制表示下的每一位。在循环到第 i 次时（从 0 开始计数），变量 a 中存储的是 a^{2^i}，若 b 该位为 1，则把此时的变量 b 累积到答案 ans 中。

值得提醒的是，在 C++语言中，两个数值执行算术运算时，以**参与运算**的最高数值类型为基准，与保存结果的变量类型无关。换言之，虽然两个 32 位整数的乘积可能超过 int 类型的表示范围，但是 CPU 只会用 1 个 32 位寄存器保存结果，造成我们常说的越界现象。因此，我们必须把**其中一个数强制转换成 64 位整数类型** long long 参与运算，从而得到正确的结果。最终对 p 取模以后，执行赋值操作时，该结果会被隐式转换成 int 存回 ans 中。

于是一个问题就出现了。因为 C++ 内置的最高整数类型是 64 位，若运算 $a*b \bmod p$ 中的三个变量 a,b,p 都在 10^{18} 级别，则不存在一个可供强制转换的 128 位整数类型，我们需要一些特殊的处理办法。

【例题】64 位整数乘法 CH0102

求 a 乘 b 对 p 取模的值，其中 $1 \le a,b,p \le 10^{18}$。

方法一

类似于快速幂的思想，把整数 b 用二进制表示，即 $b = c_{k-1}2^{k-1} + c_{k-2}2^{k-2} + \cdots + c_0 2^0$，那么 $a*b = c_{k-1}*a*2^{k-1} + c_{k-2}*a*2^{k-2} + \cdots + c_0 * a * 2^0$。

因为 $a*2^i = (a*2^{i-1})*2$，若已求出 $a*2^{i-1} \bmod p$，则计算 $(a*2^{i-1})*2 \bmod p$ 时，运算过程中每一步的结果都不超过 $2*10^{18}$，仍然在 64 位整数 long long 的表示范围内，所以很容易通过 k 次递推求出每个乘积项。当 $c_i = 1$ 时，把该乘积项累加到答案中即可。时间复杂度为 $O(\log_2 b)$。

```
long long mul(long long a, long long b, long long p) {
    long long ans = 0;
    for (; b; b >>= 1) {
        if (b & 1) ans = (ans + a) % p;
        a = a * 2 % p;
    }
    return ans;
}
```

方法二

利用 $a*b \bmod p = a*b - \lfloor a*b/p \rfloor * p$，其中 $\lfloor\ \rfloor$ 表示下取整。

首先，当 $a,b<p$ 时，$a*b/p$ 下取整以后一定也小于 p。我们可以用浮点数执行 $a*b/p$ 的运算，而不用关心小数点之后的部分。浮点类型 long double 在十进制下的有效数字有 18~19 位，足够胜任。当浮点数的精度不足以保存精确数值时，它会像科学计数法一样舍弃低位，正好符合我们的要求。

另外，虽然 $a*b$ 和 $\lfloor a*b/p \rfloor * p$ 可能很大，但是二者的差一定在 $0\sim p-1$ 之间，我们只关心它们较低的若干位即可。所以，我们可以用 long long 来保存 $a*b$ 和 $\lfloor a*b/p \rfloor * p$ 各自的结果。整数运算溢出相当于舍弃高位，也正好符合我们的要求。

```
long long mul(long long a, long long b, long long p) {
    a %= p, b %= p;  // 当 a,b 一定在 0~p 之间时，此行不必要
    long long c = (long double)a * b / p;
    long long ans = a * b - c * p;
    if (ans < 0) ans += p;
    else if (ans >= p) ans -= p;
    return ans;
}
```

◆ 二进制状态压缩

二进制状态压缩，是指将一个长度为 m 的 bool 数组用一个 m 位二进制整数表示并存储的方法。利用下列位运算操作可以实现原 bool 数组中对应下标元素的存取。

操作	运算
取出整数 n 在二进制表示下的第 k 位	$(n \gg k)\ \&\ 1$
取出整数 n 在二进制表示下的第 $0\sim k-1$ 位（后 k 位）	$n\ \&\ ((1 \ll k)-1)$
把整数 n 在二进制表示下的第 k 位取反	n xor $(1 \ll k)$
对整数 n 在二进制表示下的第 k 位赋值 1	$n\ |\ (1 \ll k)$
对整数 n 在二进制表示下的第 k 位赋值 0	$n\ \&\ (\sim(1 \ll k))$

这种方法运算简便，并且节省了程序运行的时间和空间。当 m 不太大时，可以直接使用一个整数类型存储。当 m 较大时，可以使用若干个整数类型（int 数组），也可以直接利用 C++ STL 为我们提供的 bitset 实现（第 0x71 节）。

【例题】最短 Hamilton 路径 CH0103

给定一张 $n(n \le 20)$ 个点的带权无向图，点从 $0\sim n-1$ 标号，求起点 0 到终点

$n-1$ 的最短 Hamilton 路径。

Hamilton 路径的定义是从 0 到 $n-1$ 不重不漏地经过每个点恰好一次。

很容易想到本题的一种“朴素”[①]做法，就是枚举 n 个点的全排列，计算路径长度取最小值，时间复杂度为 $O(n*n!)$，使用下面的二进制状态压缩 DP 可以优化到 $O(n^2*2^n)$。关于状态压缩动态规划，我们将在第 0x56 节详细讲解。

在任意时刻如何表示哪些点已经被经过，哪些点没有被经过？可以使用一个 n 位二进制数，若其第 i 位 $(0 \le i < n)$ 为 1，则表示第 i 个点已经被经过，反之未被经过。在任意时刻还需要知道当前所处的位置，因此我们可以使用 $F[i,j]\ (0 \le i < 2^n, 0 \le j < n)$ 表示“点被经过的状态”对应的二进制数为 i，且目前处于点 j 时的最短路径。

在起点时，有 $F[1,0]=0$，即只经过了点 0（i 只有第 0 位为 1），目前处于起点 0，最短路长度为 0。方便起见，我们将 F 数组其他的值设为无穷大。最终目标是 $F[(1 \ll n)-1, n-1]$，即经过所有点（i 的所有位都是 1），处于终点 $n-1$ 的最短路。

在任意时刻，有公式 $F[i,j]=\min\{F[i \text{ xor } (1 \ll j), k]+\text{weight}(k,j)\}$，其中 $0 \le k < n$ 并且 $((i \gg j)\ \&\ 1)=1$，即当前时刻“被经过的点的状态”对应的二进制数为 i，处于点 j。因为 j 只能被恰好经过一次，所以一定是刚刚经过的，故在上一时刻“被经过的点的状态”对应的二进制数的第 j 位应该赋值为 0，也就是 $i \text{ xor } (1 \ll j)$。另外，上一时刻所处的位置可能是 $i \text{ xor } (1 \ll j)$ 中任意一个是 1 的数位 k，从 k 走到 j 需经过 $\text{weight}(k,j)$ 的路程，可以考虑所有这样的 k 取最小值。这就是该公式的由来。

```
int f[1 << 20][20];
int hamilton(int n, int weight[20][20]) {
    memset(f, 0x3f, sizeof(f));
    f[1][0] = 0;
    for (int i = 1; i < 1 << n; i++)
        for (int j = 0; j < n; j++) if (i >> j & 1)
            for (int k = 0; k < n; k++) if ((i^1<<j) >> k & 1)
                f[i][j] = min(f[i][j], f[i^1<<j][k]+weight[k][j]);
    return f[(1 << n) - 1][n - 1];
}
```

提醒：一些运算符优先级从高到低的顺序如下页表所示。最需要注意的地方是：大小关系比较的符号优先于“位与”“异或”“位或”运算。在程序实现时，如果不确定优先级，建议加括号保证运算顺序的正确性。

① 朴素算法，英文术语为 brute-force，也可直译为“暴力”求解。