世界大学生程序设计竞赛（ACM/ICPC）高级教程

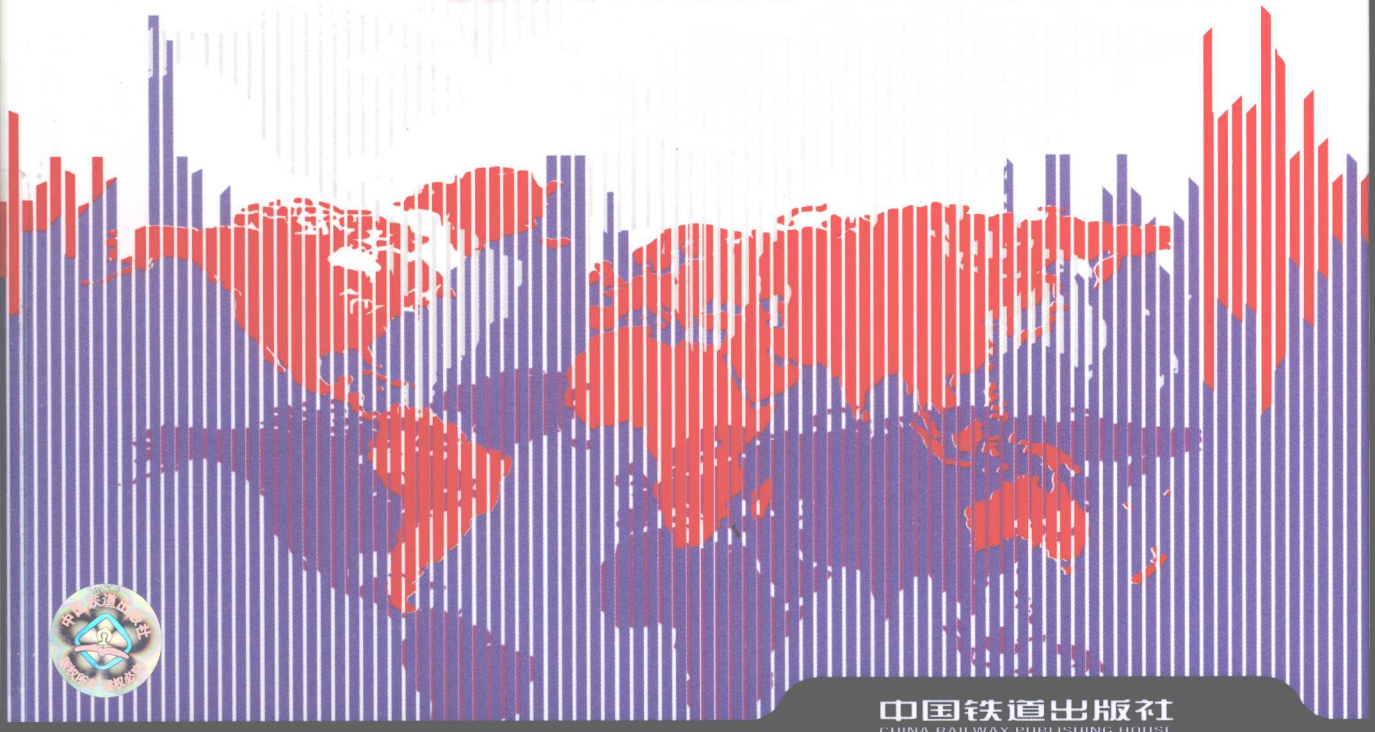
第一册

程序设计中常用的计算思维方式

吴文虎　王建德　编著

ACM International Collegiate Programming Contest Advanced Course

The Common Mode of Computational Thinking in Program



中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

世界大学生程序设计竞赛（ACM/ICPC）高级教程

第一册

程序设计中常用的计算思维方式

吴文虎　王建德　编著

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

内 容 简 介

本书是针对世界大学生程序设计竞赛（ACM/ICPC）而编写的参考书。

ACM/ICPC 是大学生的智力与计算机解题能力的竞赛，是世界公认的最具影响力的、规模最大的国际顶级赛事，被称为大学生的信息学奥林匹克。

计算机解题的核心是算法设计，而算法设计需要具备良好的数学素养。数学具有运用抽象思维去把握实际的能力，应用数学知识去解决实际问题时的建模过程是一个突出主要因素的科学抽象过程。进行抽象和形式化需要学习和掌握常用的计算思维方式。本书主要介绍了大赛程序设计中的常用思维方式，主要包括正确认识和处理整体与部分的关系、构造性思维、目标转化的思想、分类与分治思想、逆向思维、猜想与试验六个章节，旨在引导参赛学生学习并掌握编程解题的一般思维方法和过程，提高解题能力。

本书面向参加世界大学生程序设计竞赛（ACM/ICPC）的高等院校学生，也可作为程序设计爱好者的参考用书。

图书在版编目（CIP）数据

世界大学生程序设计竞赛(ACM/ICPC)高级教程. 第一册，程序设计中常用的计算思维方式/吴文虎，王建德编著. —北京：中国铁道出版社，2009.6
ISBN 978-7-113-10134-3

Ⅰ.世… Ⅱ.①吴…②王… Ⅲ.程序设计－竞赛－高等学校－自学参考资料 Ⅳ.TP311.1

中国版本图书馆 CIP 数据核字（2009）第 089247 号

书　　名：世界大学生程序设计竞赛（ACM/ICPC）高级教程　第一册　程序设计中常用的计算思维方式
作　　者：吴文虎　王建德　编著

策划编辑：严晓舟
责任编辑：秦绪好　　**编辑部电话：**（010）63583215
编辑助理：李　旸　邱雪姣　　**封面制作：**白　雪
封面设计：付　巍　　**责任印制：**李　佳

出版发行：中国铁道出版社（北京市宣武区右安门西街 8 号　　邮政编码：100054）
印　　刷：北京市兴顺印刷厂
版　　次：2009 年 7 月第 1 版　　2009 年 7 月第 1 次印刷
开　　本：787mm×1092mm　1/16　印张：18　字数：420 千
印　　数：4 000 册
书　　号：ISBN 978-7-113-10134-3/TP・3344
定　　价：42.00 元

前 言
FOREWORD

ACM/ICPC 是国际计算机协会（association for computing machinery）组织的国际大学生程序设计竞赛（international collegiate programming contest）的英文简称。这项每年一届的计算机学科竞赛始于 1976 年，是目前规模最大且最具影响力的全球性高校之间的赛事。

每年度的 ACM/ICPC 赛事从当年的 9 月份开始，先进行各大洲各地区的预选赛，从上千所高校的几千支队伍中挑选出几十支优胜队伍。让这些百里挑一的队伍在下一年春天参加总决赛，争夺金银铜奖和世界冠军的奖杯。参赛选手由三人组成一队共用一台计算机，所以这项赛事与中学生的信息学奥林匹克竞赛既有联系又有较大区别，被称为大学生的信息学奥林匹克。

2008—2009 年度的 ACM/ICPC 是第 33 届赛事，有 1 838 所大学的 7 109 支队伍参加分区赛。经过第一阶段的预选赛，共有 100 支队伍取得决赛资格，于 2009 年 4 月 18—22 日在瑞典斯德哥尔摩举行全球总决赛。

ACM/ICPC 这项国际顶级赛事是大学生智力与计算机解题能力的竞赛，是大学生展示水平与才华的大舞台，是著名的高等学府计算机教育成果的直接体现，也是 IT 企业与世界顶尖计算机人才对话的最佳机会。因而，ACM/ICPC 吸引了越来越多的高校参赛，使得参赛队伍的水平上升很快，赛题的难度也在不断提高。

能够参加 ACM/ICPC 的选手需要具备很强的数学建模功底、广博的算法知识和超强的编程能力，以及团队的合作与协同能力。ACM/ICPC 的胜负是答对题目的数量多者占优，当两个队解题数量相同的情况下，总用时最少者占优，因此解题速度非常关键。如果比赛一开始就能迅速找出竞赛中相对简单的题目并尽快加以解决，队伍的成绩排名就会占有优势，心理上的压力也会小些。相反，一开始就没有选好题，或者所写的程序总有这样或那样的错误，要花很多时间去调试排错，就会浪费宝贵的时间，处于下风。

在这种你追我赶的激烈的赛场上，比的是谁做得又快又好。竞赛过程中第一个重要的环节是看题、审题和选题。一开始就选对题，一下子就切

入主题是十分重要的。有时第一个环节遇到陷阱，“马失前蹄”，就会导致一筹莫展而步步落后。这取决于实践能力和洞察力。而实践能力与洞察力的提升需要实战，需要经验，需要学懂计算思维方式和解题策略。

众所周知，计算机解题的核心是算法设计，而算法设计需要具备良好的数学素养。数学具有运用抽象思维去把握实在的能力，应用数学知识去解决实际问题时的建模过程是一个突出主要因素的科学抽象过程。进行抽象和形式化需要学习和掌握常用的计算思维方式。

参加 ACM/ICPC 活动，在与编程高手过招的过程中可以把知识运用的综合性、灵活性和探索性发挥到极致，体验和感受数学思维与算法艺术之美，提升科学思维能力。

科学思维能力的提高是成就事业的最重要的一个因素，我们希望这本书能够对读者起到帮助作用。

王建德老师与我愉快地合作了 20 年。在本书的策划和写作中，王建德老师一如既往地花费了很多心血，总结出十分精彩的观点和思路。

清华大学的一些选手曾试用和验证过原稿中的某些算法，邬晓钧博士和徐明星博士对原稿提出过宝贵意见，在此一并感谢。

清华大学计算机系教授，博士生导师
原国际信息学奥林匹克中国队总教练

吴文虎

2009年5月

本书导学
ABSTRACT

编程解题的一般思维方法或过程，可以概述为“观察—联想—变换”，即通过对问题的观察，认识和理解该问题；然后通过联想，寻找该问题同已有知识和经验之间的联系；最后通过变换，把该问题转化为另一个或几个易于解决的新问题，最终达到解决原问题的目的。

“观察”是人类认识客观事物的基本途径，就编程解题而言，“观察”是“联想”和“变换”的基础。一般地说，通过观察应当明确：求解的对象是什么；是枚举方案还是回答哪个存在性问题；已知的条件(包括隐含条件)是什么；能否用递推公式、递归公式、约束规则或状态转移方程把问题的条件、结论和求解途径表示出来；问题所涉及的这些计算式子各有什么特点；等等。

“联想”是由某种对象而引出其他相关对象的思维形式。就编程解题而言，“联想”的目的在于为“变换”提供可能的方向或线索。一般地说，在“观察”的基础上，通过联想应当明确：以前是否解过这类试题；是否解答过与其类似而又稍有不同的试题；是否解答过与其有关的问题；能否利用解答这些问题时所使用的解题方法或所得到的结果；能否回忆出某个可能用得上的定理、公式或解题思路；为了能利用它，是否应当改变条件或结论的表现形式；等等。

“变换”是编程解题的基本手段。在“观察”和“联想”的基础上，有目的地对问题实施“变换”，把原问题转化为另一个或几个易于解决的新问题，这是编程解题成功的关键。为此，变换时，应当遵循如下三条基本的思考原则：熟悉化原则、简单化原则以及和谐化原则。

为了使读者对“观察—联想—变换”的思维方法和过程有一个比较全面深入的了解，我们归纳了六种常用的思维方法。

在“观察”上，我们提出了整体与部分的思想，包括：

(1) 整体实现的关键是准确地应用必要条件。

(2) 整体思考的一个重要角度是“守恒”，即寻找变化中的不变量。

(3) 提高整体实现效率的途径是“充分利用有效信息”和“压缩冗余信息”。

(4) 改善整体的性能状态的基础是处理好细节问题。

在“联想”上，我们提出了逆向思维和猜想与试验，分析了“执果索因型”的逆向思维和“由反及正型”的逆向思维；探讨了四种联想方式：相似联想、归纳联想、从数与形的结合上联想和“回到起点”重新联想。指出猜想是在深入分析问题的基础上，不懈探索、反复修正的过程。

在“变换”上，我们提出了构造性思维、目标转化思想、分类与分治思想。构造性思维包括建立模型的机理分析法和统计分析法；建模过程注意应用序关系；选择模型时必须权衡四个因素：“时间复杂度、空间复杂度、编程复杂度和思维复杂度”。目标转化思想包括缩小目标的“降维”思想和放大目标的“升维”思想；分类与分治思想包括应用于一般有序序列的“二分查找”；应用于退化了的有序序列的“二分枚举”；应用于无序序列的“二分搜索”；应用于多维情况的“多重二分”。

在实际编程解题时，“观察”、“联想”、“变换”等思想活动总是互相联系、互相影响、互相交织地进行着，形成了一个有机的整体。本书列举的六种思维方式是互相渗透的，章节划分主要是依据各种思维方式的主要特征进行分类，同时也是为了叙述的方便。当然这六种思维方式并没有、也不可能穷尽编程解题过程中的所有思维活动，它只不过是列举了常用的一些思维方式，为“观察—联想—变换”的思维活动勾勒出一个基本轮廓，为读者留下学习、探索和再创造的空间。

目 录
CONTENTS

第1章 正确认识和处理整体与部分的关系

“整体”与“部分”是一对虽然对立、但并非僵化不变的概念。在一定条件下，“部分”可以看做“整体”，“整体”又可以看做是另一个“整体”的“部分”，两者互相依存和影响。“整体”与“部分”又是可以互相转化的。例如，冗余信息相对全部信息而言，是“部分”；细节处理和设计测试数据相对于编程全部过程和可能解集而言，也是“部分”。但如果处理不好，同样会导致效率过低或程序出错，影响整体的实现。“整体”的问题可以分割成“部分”来处理，“部分”的问题也可通过“整体”来解决。因此，正确运用辩证思想，认识和处理“整体”与“部分”的关系，对提高解题能力是十分重要的。

本章将从以下四个角度讨论整体与部分的关系：

① 整体实现的关键是准确地应用必要条件。

② 整体思考的一个重要角度是“守恒”。

③ 提高整体实现效率的途径是“充分利用有效信息”和“压缩冗余信息”。

④ 改善整体的性能状态的基础是处理好细节问题。

1.1 整体实现的关键是准确地应用必要条件

数学上，判断真假的语句叫做命题。常用小写拉丁字母 p，q，r，s…来表示。如果由命题 p 经过推理可以得出命题 q，也就是说，“如果 p 成立，那么 q 成立”，则记为 $p \Rightarrow q$。

一般地，如果已知 $p \Rightarrow q$，那么就说，p 是 q 的充分条件，q 是 p 的必要条件。例如，$x^2>0$ 是 $x>0$ 的必要条件。在这里，$x^2>0$ 的解集包含了 $x>0$ 的解集（$\{x \mid x>0\} \in \{x \mid x^2>0\}$），可用图 1-1 中的文氏图表示。

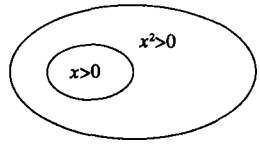


图 1-1 $x^2>0$ 的解集包含 $x>0$ 解集的文氏图

由此可见，必要条件是命题之间的一种逻辑关系。准确地应用必要条件，有助于揭示问题的本质或简化原有模型，从而找到

高效的解决方法。

1.1.1 选择有助于简化问题、变难为易的必要条件

大部分问题所要求解的往往是符合某一些条件的方案或具体数值。将这些条件命题记为 p，其解集用 A 表示。在解题过程中，不可能在一开始就知道 A 的确切解，而是从已知条件 q 出发，运用枚举、搜索、动态规划、贪心等方法，在一个较大的可能解集 B 内进行筛选，最终找到符合 p 的解集 A。在这一过程中，B 包括了 A，A 也必定符合条件 q，有 $p \Rightarrow q$，q 是命题 p 的必要条件。从比较“宽松”的条件 q 入手，在 B 中寻找 A，使其满足比较“苛刻”的条件 p。

可以看到，开始界定的必要条件 q 的“宽松”程度，即 B 的范围大小直接影响着解决问题的效率。正如“混合物中真金含量愈高，需要清除的杂质愈少”的自然法则一样，q 愈趋近于 p，则 B 愈趋近于 A，求解范围自然愈小。当然，这必须建立在可行的基础之上。因为如果条件 q 在判断处理时难于实现，或是要花费大量时间，那么无论 q 再怎么精确，都是毫无意义的。

于是，这就启发我们要在坚持“简化问题、变难为易”的原则下，尽力寻找“精确”的必要条件，以缩小求解范围，提高出解速度。

【例题1.1】寻找完备匹配

已知二分图 $G(V,E)$，求哪些边是构成完备匹配所必需的。共 $2n$ 个顶点，$n \leqslant 100$。

思路点拨

设构成完备匹配所必需的边的集合为 A，当且仅当某边 e 被去掉后，子图 g 不存在完备匹配，则有 $e \in A$。这可算是条件命题 p。思考到这里，可以得到：

算法 1：枚举所有边

枚举图 G 中所有边 E：对于某边 e，在原图中去掉，$g=G-e$，对 g 求最大匹配。若不存在完备匹配，则表示 $e \in A$，否则反之。

用匈牙利算法求最大匹配，时间复杂度是 $O(E*n)$，枚举所有边需 $O(n^2)$。于是此算法时间复杂度为 $O(E*n^3)$，过于庞大。

在上述算法中，按照常规思路，一开始就不经意地把初始范围 B 设成全题边集 E，必要条件 q 为“满足是 E 中的边即可”。这样，便把求解的范围定大了。其实，若边 $e \in A$，那么 e 必属于任意一个完备匹配，这是显然的。换句话说，就是任意一个完备匹配都包含了所有必需的边集 A。于是，得到了算法 2。

算法 2：枚举完备匹配 B 中的部分边

先求出 G 中的一个完备匹配，设为 B。枚举 B 中所有边 E：对于某边 e，在原图中去掉，$g=G-e$，对 g 求最大匹配。若不存在完备匹配，则表示 $e \in A$，否则反之。

在算法 2 中，实际上是把初始范围 B 缩小为一个完备匹配，必要条件 q 精确为“边属于完备匹配 B”。这么一来，时间复杂度下降为 $O((E+h^2)*n)$（h 为完备匹配 B 的边数）。下面给出算法流程：

```
var
   n,i,e:longint;                          /*顶点数为 2*n，边数为 e*/
   used,vx:array[1..200]of boolean;        /*顶点 i 的访问标志为 used[i];增广轨标志为 vx[i]*/
   a:array[1..200,1..2]of longint;         /*第 i 条边为(a[i,1],a[i,2])*/
   match:array[1..200]of longint;          /*与顶点 i 相连的匹配边为(i,match[i])*/
   g:array[1..200,1..200]of longint;       /*二分图*/
func find(v:integer):boolean;              /*搜索以 v 顶点出发的可增广轨。若存在，则返回
                                           true，否则返回 false*/
var j:longint;
{  find←true;
   for j←1 to n*2 do                       /*搜索所有与 v 顶点相连的未访问点*/
     if(g[v,j]=1)and(not used[j])
       then{ used[j]←true;                 /*置顶点 j 访问标志*/
             if match[j]=0   /*若 j 为未盖点，则 v 进入可增广轨，(v,j)为匹配边，成功退出*/
               then {vx[v]←true;match[j]←v;exit}/*then*/
               else if find(match[j]) then {vx[v]←true;match[j]←v;exit};
          /*若从盖点 j 相连的匹配边的另一端点出发，存在可增广轨，则 v 进入可增广轨，(v,j)
          为匹配边，成功退出*/
            };                             /*then*/
   find←false;                             /*失败退出*/
};                                         /*find*/
func edmonds:longint;                      /*计算当前二分图的最大匹配边数*/
var m,i:longint;
{  fillchar(used,sizeof(used),false);      /*所有顶点未访问*/
   fillchar(vx,sizeof(vx),false);          /*所有顶点未在可增广轨上*/
   fillchar(match,sizeof(match),0);        /*所有顶点为未盖点*/
   m←0;
   for i←1 to n*2 do                       /*将每条自反边设为匹配边，并进入可增广轨*/
     if g[i,i]=1 then {match[i]←i;inc(m);vx[i]←true};/*then*/
   for i←1 to n*2 do                       /*搜索每一个顶点*/
   {  fillchar(used,sizeof(used),false);/*所有顶点未访问*/
     if(not vx[i])and(find(i))then inc(m)};    /*若目前顶点 j 未在可增广轨上，但存在以
                                                该顶点出发的可增广轨，则匹配边数+1*/
   edmonds←m;                              /*返回当前二分图的最大匹配边数*/
};                                         /*edmonds*/

{  fillchar(g,sizeof(g),0);                /*初始时二分图为空*/
   read(n,e);                              /*读入二分图的顶点数和边数*/
   for i←1 to e do                         /*读入边信息，构造二分图*/
   {  read(a[i,1],a[i,2]);g[a[i,1],a[i,2]]←1};/*for*/
   for i←1 to e do                         /*枚举所有边*/
   {  g[a[i,1],a[i,2]]←0;                  /*撤去第 i 条边*/
     if edmonds<n then writeln(i);         /*若剩余图不存在完备匹配，则第 i 条边属于必
                                           需边的集合，输出该边*/
     g[a[i,1],a[i,2]]←1;                   /*恢复第 i 条边*/
   };                                      /*for*/
}.                                         /*main*/
```

例题 1.1 中 B 集合的确定，充分体现了“矛盾的普遍性寓于特殊性之中，并通过特殊性表现出来”的哲学原理，这一道理和解题过程中的思维习惯其实是一致的。当碰到一道难题时，总是尝试从最简单的特殊情况入手，找出有助于简化问题、变难为易的必要条件，逐渐深入，最终分析归纳出一般规律。

1.1.2 合成必要条件，从整体结构上优化

前面已经看到，利用必要条件有助于解决存在性问题。其实，在搜索和动态规划中，必要条件也有其应用价值。一般地，如果深度优先搜索、广度优先搜索采取完全盲目的搜索方式，则时间复杂度往往是指数级的，一般难以承受。于是，如何限制搜索范围、减少搜索量就变成了解题的关键。其中的一个有效手段就是“剪枝”，即使用必要条件剪去一些不必要的搜索分支。由于问题的错综复杂性，单一剪枝的力度往往不够。当内在因素相互影响而难以总体把握的时候，可以尝试从多个侧面分析寻找必要条件，把问题分解，根据各部分的本质联系，将各方面的必要条件综合起来使用，往往能起到一种合力的效果。

【例题1.2】移棋子

有一个 5×5 的方格棋盘，棋盘上放着 24 颗不同的棋子，分别用英文大写字母 A，B，…，X 来表示，棋盘上还有一个方格空着，用空格表示。

游戏的每一步是将空格上方、下方、左方或是右方的棋子移入空格，这四种操作分别用 1、2、3、4 来表示。

如果给出棋盘的初始状态和一定顺序的有限操作序列，就可以得到唯一的目标状态。例如，图 1-2 中的初态经过操作序列“144223”到达终态。

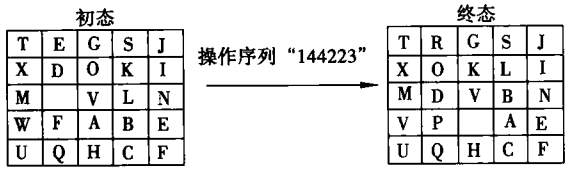


图 1-2 棋盘的初始状态（左图）经过操作后得到目标状态（右图）

但是，原来正确的操作序列顺序被打乱了，初态按照被打乱的操作序列并不能得到终态（仅仅是顺序打乱了，各类型操作总数不变）。

现已知棋盘的初态、棋盘的终态和被打乱后的操作序列（操作序列长度 $L \leqslant 50$），要求计算和输出原来正确的操作序列。若无解，则输出“0”。

思路点拨

这是一道典型的搜索题。根据题意可得如下信息：

① 在有限步内把原始状态转移至目标状态，或确定无解。

② 移动步数已知。

③ 各个移动方向的步数已知，但序列来知。

④ 解必须满足②和③限制。

采用深度优先搜索，逐一产生各种可能的移动序列，输出满足题设的即可。显然，这种完全盲目的搜索在时间上令人无法忍受，此时可考虑从以下几方面来剪枝：

① 移动方向的数目。由于各移动方向的数目是已知的，因此一旦某个移动方向的个数超出限制，则可不搜索此方向。

② 空格的位置和各方向的数目。由于目前状态和目标状态都已知，各方向的数目也已知，因此可确定目前状态的空格能否由此方向集合移至目标状态位置。即必须满足

（目标状态空格横坐标-当前空格横坐标=方向 4 所剩数目-方向 3 所剩数目）and（目标状态空格纵坐标-当前空格纵坐标=方向 2 所剩数目-方向 1 所剩数目）

③ 各字符的位置和各方向数目。同剪枝条件②一样，可以根据目前状态和目标状态以及各方向的数目，确定目前状态的各字符能否由此方向集合移至目标状态位置。

在上述思考过程中，将 p 定义为“操作序列能使初态达到终态”，满足 p 的所有操作序列集合设为 A，q_1、q_2、q_3代表三项剪枝，分别满足三项剪枝的操作序列集合设为 B_1、B_2、B_3。有 $p \Rightarrow q_1$，$p \Rightarrow q_2$，$p \Rightarrow q_3$，$A \in B_1 \cap B_2 \cap B_3$。其文氏图如图 1-3 所示。

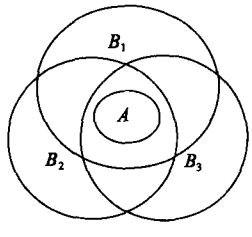


图 1-3　$A \in B_1 \cap B_2 \cap B_3$的文氏图

这里，共运用了三个必要条件。从图中可以看出，三管齐下使求解范围缩得更小、更精确。当然，要注意的是，这三项剪枝都容易实现，且没有耗费太多的时间，否则就得不偿失了。通过这些剪枝，搜索量大为减少，程序速度明显提高。下面给出算法流程：

```
const
  mx:array[1..4]of longint=(-1,1,0,0); /*四个方向上的水平增量和垂直增量*/
  my:array[1..4]of longint=(0,0,-1,1);
var
  a:array[1..2,1..5,1..5]of longint;   /*初态棋盘中（i,j）的数字为 a[1,i,j];
                                         终态棋盘中（i,j）的数字为 a[2,i,j]*/,
  x1,y1,x2,y2:array[0..24]of longint;  /*初态中数字 i 的位置为（x1[i],y1[i]);
                                         终态中数字 i 的位置为（x2[i],y2[i]）*/
  b:array[1..4]of longint;             /*第 i 种操作的数目为 b[i]*/
  c:array[1..500]of longint;           /*操作序列*/
  i,j:longint;
proc solve(n:longint);                 /*目前已确定了前 n 个操作。递归搜索第 n+1 个操作数*/
var
  i,j,m,x0,y0:longint;
  tf:boolean;
{ if n=b[0]                            /*若进行完所有操作，则检查棋盘是否到达目标状态*/
    then { tf←false;
           for i←1 to n do
             for j←1 to n do if a[1,i,j]<>a[2,i,j] then tf←true;
```

```
            if not tf                    /*若棋盘为目标状态，则输出操作序列，成功退出*/
              then {for i←1 to b[0] do write(c[i]);writeln;halt};/*then*/
        }                                /*then*/
 else { for i←1 to 4 do                  /*枚举每个方向上的操作*/
        { if(b[i]>0)and(x1[0]+mx[i]<6)and(x1[0]+mx[i]>0)and(y1[0]+my[i]<6) and
          (y1[0]+my[i]>0)/*若i方向尚需操作，且空格在棋盘内移动，则i方向进行一次操作*/
            then { dec(b[i]);x0←x1[0]+mx[i];y0←y1[0]+my[i];m←a[1,x0,y0];
                   a[1,x1[0],y1[0]]←a[1,x0,y0];a[1,x0,y0]←0;
                   x1[0]←x0;y1[0]←y0;x1[m]←x0-mx[i];y1[m]←y0-my[i];
                   c[n+1]←i;             /*确定第n+1个操作数为i*/
                   solve(n+1);           /*从第n+1个操作数出发继续递归*/
                   x1[0]←x1[m];y1[0]←y1[m]    /*恢复操作前的状态*/
                   x1[m]←x0;y1[m]←y0;a[1,x0,y0]←m;a[1,x1[0],y1[0]]←0;
                 };                      /*then*/
        };                               /*for*/
      };                                 /*else*/
};                                       /*solve*/
{ for i←1 to 5 do       /*读入初态棋盘中每一个格子的数字，计算每一个数字的初始位置*/
  { for j←1 to 5 do {read(a[1,i,j]);x1[a[1,i,j]]←i;y1[a[1,i,j]]←j};
    readln;
  };                                     /*for*/
  for i←1 to 5 do       /*读入终态棋盘中每一个格子的数字，计算每一个数字的目标位置*/
  { for j←1 to 5 do {read(a[2,i,j]);x2[a[2,i,j]]←i;y2[a[2,i,j]]←j}; /*for*/
    readln;
  };                                     /*for*/
  readln(b[1],b[2],b[3],b[4]);           /*读每种操作的次数*/
  b[0]←b[1]+b[2]+b[3]+b[4];              /*累计操作的总次数*/
  if(x2[0]-x1[0]=b[2]-b[1])and(y2[0]-y1[0]=b[4]-b[3])then solve(0);/*若水平
                移动方向和垂直移动方向的个数满足限制，则从空的操作序列出发，递归搜索*/
  writeln(0);
}.                                       /*main*/
```

必要条件在动态规划方面也有应用，不妨看看如下的一个实例。

【例题1.3】中世纪剑客

在路易十三和红衣主教黎塞留当权的时代，发生了一场决斗。n 个人站成一个圈，依次抽签。抽中的人和在他右边的人决斗，负者出圈。这场决斗的最终结果关键取决于决斗的顺序。现已知任意两人决斗中谁能胜出的信息，但“A 赢了 B”的这种关系是没有传递性的。例如，A 比 B 强，B 比 C 强，C 比 A 强。如果 A 和 B 先决斗，C 最终会赢；但如果 B 和 C 决斗在先，则最后 A 会赢。显然，他们三人中的第一场决斗直接影响最终结果。

任务：n 个人围成一圈，按顺序编上号 1～n。一共进行了 n-1 场决斗。第一场，其中一人（设 i 号）和他右边的人（即 i+1 号，若 i=n，其右边的人则为 1 号）。负者被淘汰出圈外，由他旁边的

人补上他的位置。已知 n 个人之间的强弱关系由关系矩阵 A 来描述，如果 A_{ij}=1，则表示第 i 人强于第 j 人；如果 A_{ij}=0，则表示第 i 人弱于第 j 人。如果存在一种抽签方式使第 k 人可能胜出，则我们说第 k 人有可能胜出。我们的任务是读入规模在 $3\leqslant n\leqslant 100$ 范围内的矩阵 A（注意，当 $i<>j$ 时，$A_{ij}=1-A_{ji}$，假设对每个 i，总有 $A_{ij}=1$），计算和输出可能胜出的人数和方案。

思路点拨

这道题满足最优子结构和重叠子问题的特征，可以采用动态规划的方法来解。

设用 $e[i,j]$记录 i 与 j 之间的胜负情况，当 i 胜 j 时，$e[i,j]$=true；当 j 胜 i 时，$e[i,j]$=false。

用 $a[i,j,k]$记录 k 是否能在 $i\sim j$ 这些连续的人之间取胜（k 在其中），肯定时为 true，反之为 false。

如果 k 在 $i\sim j$ 这些连续的人之间取胜，则 k 有位于左子序列$[i,u-1]$和位于右子序列$[x+1,j]$的两种情况，如图 1-4 所示。

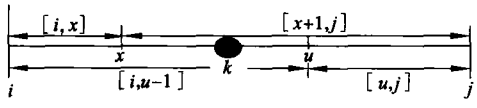


图 1-4　k 在$[i,j]$取胜的两种情况

根据这两种情况，得出两个必要条件：

① k 在左子序列$[i,u-1]$胜出，则必须战胜右子序列$[u,j]$的胜者 v（$k<u\leqslant v\leqslant j$）。

② k 在右子序列$[x+1,j]$胜出，则必须战胜左子序列$[i,x]$的胜者 y（$i\leqslant y\leqslant x<k$）。

按照图 1-4，写出如下六个记录能够胜出的表示式：

① k 在左子序列$[i,u-1]$胜出可表示为 $a[i,u-1,k]$。

② v 在右子序列$[u,j]$胜出可表示为 $a[u,j,v]$。

③ k 在右子序列$[x+1,j]$胜出可表示为 $a[x+1,j,k]$。

④ y 在左子序列$[i,u-1]$胜出可表示为 $a[i,x,y]$。

⑤ k 胜 v 可表示为 $e[k,v]$。

⑥ k 胜 y 可表示为 $e[k,y]$。

由此可得出状态转移方程

$$a[i,j,k]=\begin{cases}\text{true} & \text{存在}(a[i,x,y]\text{ and }a[x+1,j,k]\text{ and }e[k,y])\text{ or }(a[u,j,v]\text{ and }a[i,u-1,k]\text{ and }e[k,v])=\text{true}\\ & (i\leqslant y\leqslant x<k,\ k<u\leqslant v\leqslant j)\\ \text{false} & \text{否则}\end{cases}$$

此算法的时间复杂度为 $O(n^5)$，空间复杂度为 $O(n^3)$，该值过大，所以必须改进。

上述解法忽略了一个情况：编号为 k 的人能从所有人中胜出，必要条件是他能与自己“相遇”，即把环看成链，将 k 点拆成两个分别在一头一尾，中间的人全部被淘汰出局，k 保持不败。这样，在连续几个人的链中，只须考虑头尾两个人能否胜利会师，中间的则不予考虑，从而少了一维状态表示量。设 meet$[i,j]$记录 i 和 j 能否相遇，能相遇则为 true，否则为 false。状态转移方程为

$$\text{meet}[i,j]=\begin{cases}\text{true} & \text{存在 meet}[i,k]\text{ and meet}[k,j]\text{ and }(e[i,k]\text{ or }e[j,k])=\text{true}\ (i<k<j)\\ \text{false} & \text{否则}\end{cases}$$