第3章 算法设计基础

算法描述解决问题的方法和途径，是程序设计的基础和精髓，采用高效算法才能设计出优质程序。算法的设计、实现及评价对于程序设计具有重要意义。

3.1 算法的描述

算法的描述具有重要意义，描述一个算法的目的是使其他人利用该算法解决具体问题。算法的描述方式没有统一规定，既可以使用自然语言方式，也可以使用类似于某种高级语言的伪代码，还可以使用程序流程图，N/S 盒图等方式。在软件开发的不同阶段，描述算法的具体目的有所不同，应针对不同目的选择适当的描述方法。例如，在软件开发初期，如分析阶段，描述算法应尽可能地反映客观现实的真实过程或抽象的数据处理过程，不要过多地考虑如何编程实现算法，也不要考虑未来程序的层次结构问题，所以可使用自然语言或伪代码方式描述；在软件开发后期，如详细设计或编码阶段，则倾向使算法逐步变换为程序代码，算法的描述应反映出最终程序的结构，因此，该阶段适合使用程序流程图、N/S 盒图等描述，以便将解决问题的重点放在如何使用某种高级语言，并以良好的编程风格高效地实现算法上。

3.1.1 自然语言方式

以自然语言方式描述的算法每一步处理都很直白，不懂程序设计语言的人也可以描述算法。但是，由于描述的风格不确定，描述的层次结构不清晰，当程序规模稍大时难以读懂，所以这种方式不适合规模较大的程序。

【例 3.1】 算法描述示例：用自然语言描述的直接选择排序(升序)算法。

算法 3-1 直接选择排序。

输入：n 个数放置在数组 a[n]中。

第 1 步　令 i=1；

第 2 步　若 i<n，执行第 3 步，否则转第 10 步；

第 3 步　令 k=i，顺次执行第 4 步；

第 4 步　令 j=i+1，顺次执行第 5 步；

第 5 步　若 j≤n，执行第 6 步，否则转第 8 步；

第 6 步　若 a[j]<a[k]，则置 k 为 j，然后顺次执行第 7 步，否则直接执行第 7 步；

第 7 步　置 j 为 j＋1，转第 5 步；

第 8 步　若 i!＝k，则交换 a[i]与 a[k]中的数，顺次执行第 9 步，否则直接执行第 9 步；

第 9 步　置 i 为 i＋1，转第 2 步；

第 10 步　算法结束。

上述算法定义了一种被称为“直接选择排序”的排序过程，将 n 个数据 a_1，a_2，…，a_{n-1}，a_n 从小到大排序。其主要思想是将未经排序数据中的最小一个挑出来，然后与未排序部分最左侧第一个未排序元素位置互换，于是已经排序的数据增加了一个，而未经排序的数据减少了一个，反复执行这个过程直到未排序数据的数目为 1，至此，所有数据都已经进行了排序。

3.1.2　程序流程图方式

程序流程图不仅能够描述程序，也能够描述算法，它使算法的流程控制非常直观地表现出来，成为软件设计人员的有力工具，如图 3.1 所示，这种图也被称为程序框图。

程序流程图中使用的符号说明如下。

(1) 开始与结束：中间平行，两边圆弧的框。

(2) 判断：菱形框。

(3) 功能框：矩形框。

(4) 输入输出：平行四边形框。

(5) 流程：直线为流程线，箭头为流向。

【例 3.2】 用程序流程图表示求函数 F(X)＝ln(1/x)＋ln(cos(x))的值(x≠0)。

算法流程图如图 3.1 所示。

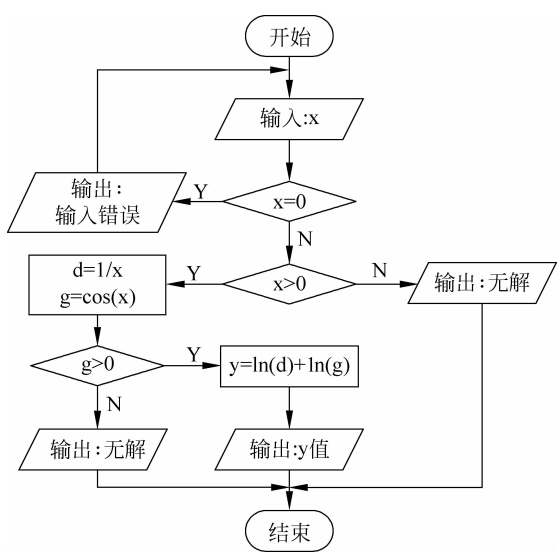


图　3.1

这种图不仅对算法控制流程的描绘十分直观，而且，根据它可以高效率地编写软件代码。但是，它仅对小型软件效果较好。

3.1.3　伪代码方式

使用伪代码(pseudo-code)描述算法的优点是：尽管风格趋向形式化，但仍然比较贴近自然语言描述，易于理解；表达方式简捷，贴近解决问题的数学运算过程；将注意力完全集中在算法的阐述上，不必去勾画任何图形；伪代码通常接近于某种程序设计语言的风格，比较容易将算法描述直接转化为程序，本书中伪代码风格与 C++程序风格类似。

在伪代码描述方式中，只允许使用具有确切含义的动词和名词，预先规定了少量的关键字。在书写时，为了表达算法的层次，常需要缩进。本书伪代码构成元素和书写规则如下。

(1) 标题行：一个算法应该开始于标题行，标题行中包括算法的编号和中文名称，例如：

算法 3-1　选择排序

如果该算法可以被其他算法调用，那么在标题行的下面一行应有英文算法名以及调用算法所需要的参数列表，例如：

算法 3-1　选择排序
Sort(A[n])

其中，英文算法名由英文字母开始，其后跟随若干字母或数字，例如 Sort1、Sort2 等。上述英文算法名中参数 A[n]表示 n 个数组成的序列(数组)。

(2) 层次：算法的书写应该具有层次，每一层中的每一条可以规定标号，下面的一层采用缩进方式，同层次的缩进相同，例如：

1. XXXXXXXX
(1)XXXXXX
(2)XXXXXXX
　　XXXXXX
　　XXXXXX
(3)XXXXXX
2. XXXXXXXXXX

(3) 注释：注释用来说明算法中某部分的功能，其形式是由[]括起来的中文或英文字符串，如例 3.3 所示。

(4) 控制结构：出现在算法中的一些操作之间具有确定的联系，但是，各操作的物理排列次序不一定是逻辑上操作的执行顺序，例如执行完一个操作以后具体执行哪个操作，需要根据这个操作的类型决定，这种操作间存在的执行顺序关系称为控制结构。通常以固定的格式描述那些相关的操作，可以使用这些格式将算法中的操作组织成为不同的层次，勾勒出算法的物理结构。下面列出描述算法时经常使用的 3 种控制结构。

① 顺序结构：表明算法中各操作间执行顺序的先后关系，这种结构中的各个操作将按照其出现顺序依次进行，例如，下面两个操作中将先执行<操作 1>，然后再执行<操作 2>。

<操作 1>
<操作 2>

② 分支结构：又称选择结构，这种结构在运行时根据给定条件是否成立，选择具体的执行路径，这种结构有以下两种格式。

格式 1：if(<条件 1>)则 <操作 1>
　　否则 <操作 2>

格式 2：if(<条件 2>)则<操作>

其中，格式 1 的语义是：如＜条件 1＞为真，则执行＜操作 1＞，然后执行紧随该结构后的操作；如＜条件 1＞为假，则执行＜操作 2＞，然后执行紧随其后的操作。格式 2 的语义是：如＜条件 2＞成立则执行＜操作＞，然后执行紧随该结构后的操作；如＜条件 2＞为假则执行紧随该结构后的操作。另外还经常使用多重选择。

多重选择是分条件执行下述操作。

```
标号 1<条件 1>,执行<语句 1>
标号 2<条件 2>,执行<语句 2>
        ⋮
标号 n<条件 n>,执行<语句 n>
```

上述多重选择的含义是：当这组条件中的某一条满足时，执行相应的语句，然后执行紧随该结构后的操作。注意：上述条件是互相排斥的。

③ 循环结构：在满足循环条件的前提下重复执行＜操作＞。

格式 1：do<操作>
　　while(<循环条件>)

格式 2：while(<循环条件>)do <操作>

格式 3：for <表达式 1> to<表达式 2> 步长 <表达式 3>
　　<操作>

其中，格式 1 的语义是反复执行＜操作＞直到循环条件为假时停止，称为直到型循环；格式 2 的语义是当＜循环条件＞为真时反复执行＜操作＞，称为当型循环；格式 3 的语义是从＜表达式 1＞到＜表达式 2＞以＜表达式 3＞为步长反复执行＜操作＞，该循环称为步长型循环。

(5) 子算法调用：调用一个已知算法的书写方式如下。

```
调用<英文算法名>(<调用参数表>)
```

(6) 需要标明算法的“开始”和“结束”点。

【例 3.3】 算法描述示例：伪代码描述的直接选择排序(升序)算法。

算法 3-1　直接选择排序

```
Sort(a[],n)
[算法开始]
[每循环一次在未排序元素中找出一个最小的元素进行排序]
for i←1 to n－1 步长为 1
(1)[准备]
      k←i
(2)[查找未排序元素中最小的元素]
      for j←i＋1 to n 步长为 1
          if(a[j]< a[k])
          则 k←j
```

```
(3)if(i! = k)[交换两个元素]
    则 x←a[i]
       a[i]←a[k]
       a[k]←x
[算法结束]
```

3.2　结构化算法设计初步

3.2.1　算法结构

随着算法规模的增大和复杂性的提高，算法的可读性变得非常重要，提高算法易读性的途径之一是按照不同的层次描述算法。研究发现，无论多么复杂的算法总可以使用顺序结构、循环结构和选择结构这 3 种基本控制结构，以及一些附加的规定将算法的层次结构清晰地描述出来，这种描述风格的算法被称为结构化算法。

1. 顺序结构

算法操作按照它在算法中出现的顺序逐条执行，如图 3.2 所示。

2. 选择结构

算法操作执行与否由某种条件的成立与否决定，即由选择条件控制，如图 3.3 所示。

3. 循环结构

循环结构又称为重复结构，它是一种控制算法多次执行某些操作的结构，如图 3.4 所示。

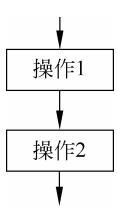


图　3.2

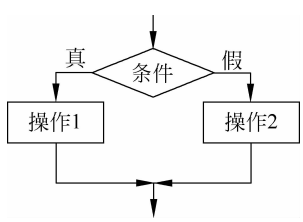


图　3.3

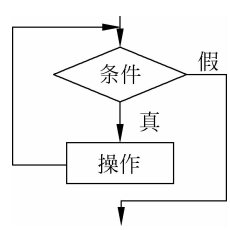


图　3.4

结构化算法的特点在于算法的层次结构，编写算法时必须遵守下述规则。

(1) 按照不同层次将组成算法的多个操作划分成一组，每组代表一种复杂的相对独立的复合操作，各复合操作的内部还可以划分层次，从而形成整个算法的嵌套层次结构。

(2) 每组操作具有唯一的入口和唯一的出口，即一端进一端出。这样的操作组置于其他操作中时，算法的执行顺序必定是从前一组操作的出口到本组操作的入口，经过本组内部的运算，到达本组操作的唯一出口。

(3) 各组之间利用顺序、选择和循环 3 种控制结构进行连接，形成更高层次的算法结构。

3.2.2 算法设计

算法设计应该遵守自顶向下、逐步求精的原则。首先，确定用户初期对问题模糊笼统的描述，提出概略的解题思路；其次，逐步细化与展开解题步骤和操作，直到算法中所有处理变得详细和确定为止，将详细确定的解题步骤和操作转化成基本运算，并用结构化算法的3种结构将其表示出来；最后，完成算法的详细设计和描述。

1. 主体结构设计

一个算法通常由5个部分构成：算法开始标志块、初始化处理模块、问题处理核心模块、善后处理模块和算法结束标志块。这5部分构成算法的主体结构，设计算法时应首先从设计算法主体结构入手，再自顶向下逐个设计模块，这样才能统筹全局，使设计过程有条不紊地进行。

2. 顺序结构设计

算法的顺序结构对应于客观世界或人类思维过程中那些前后相继、循序进行的发展环节或阶段，例如，一个人一天的行为可以从整体上归结为：起床、洗漱、早餐、工作、晚餐、入寝等。顺序结构是一种最简单的线性结构，其特点是处于这种结构中的每个由若干操作组成的操作块，按照其出现的先后顺序依次执行。顺序结构是一种描述客观世界中顺序现象的重要手段。

【例3.4】 求底面半径为r，高度为h的圆柱体的侧面积和体积。

问题分析：问题分析的任务是确定问题的需求，并建立问题的数学模型，即明确问题需要做什么，并用数学语言描述该问题。本问题的要求是：首先输入圆柱体的底面半径和高，然后计算圆柱体的侧面积和体积，最后输出圆柱体的侧面积和体积。根据问题要求确定：该问题的数学模型就是计算圆柱体侧面积和体积的公式。

数据结构：r，Peri_bottom，S_bottom，S_side，V，h均为浮点型变量，分别存储圆柱体的底面半径、底面周长、底面面积、侧面面积、圆柱体积和高的值。

算法流程图：根据计算圆柱体侧面积和体积的公式，计算圆柱体的侧面积，首先应该计算圆柱体的底面周长，计算圆柱体的体积，首先应该计算圆柱的底面积，因此，设计此问题的算法如图3.5所示。算法结构为从上至下顺序排列的模块，每个模块的功能相对独立，由一些操作组成，算法的执行顺序显然就是上述模块的物理排列顺序。

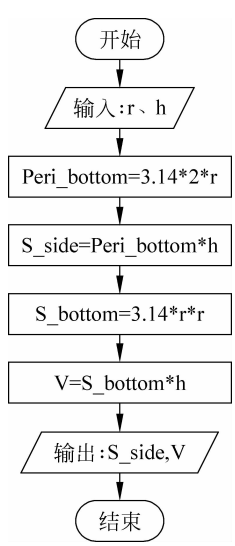


图 3.5

值得注意的是，顺序结构中每个模块内部都可以含有多个操作，这些操作之间也可能具有某种联系，从而在每个模块内部形成子结构，这些子结构也可能是选择结构或循环结构，即使这样，算法的整体结构仍然是顺序结构。

3. 选择结构设计

选择结构用于描述分支现象。分支现象广泛存在于自然界和人

类思维过程中，比如，经过每一个交叉路口时都需要选择一条岔路，目的地不同选择的岔路就有可能不同，显然，岔路的选择是以目的地为条件的。同样，人们思维时也会经常做出“如果……，则这样，否则那样”之类的推理判断。虽然具体的分支现象千差万别，但是，可以将它们分类为双分支结构与多分支结构。

1）双分支选择结构设计

双分支选择结构是最常用的选择结构，其语义很简单，即如果判断条件被满足则执行相应的处理，否则执行另外的处理。选择结构的各分支可以用“真”与“假”、“是”与“否”、“则”与“否则”等标注。

【例 3.5】 求解一元二次方程 $ax^2+bx+c=0$ 的实根。

问题分析：本问题的要求是：首先，输入一组一元二次方程系数 a、b、c 的值；其次，计算这组系数确定的一元二次方程的根；最后，输出一元二次方程的根。根据问题要求确定：该问题的数学模型就是计算一元二次方程根的公式。

数据结构：a、b、c 均为浮点型变量，分别存储方程的 3 个系数；x1 和 x2 为浮点型变量，分别存储方程的两个实根；q 为浮点型变量，存储一元二次方程判别式的值。

算法流程图：一元二次方程实数解具有一个普遍的形式，即$(-b\pm\sqrt{b^2-4ac})/2a$，该公式成立的条件是：$a\neq0$ 并且 $b^2-4ac\geqslant0$，因此，在算法描述中应对其进行判断。算法的具体描述如图 3.6 所示。算法的主体结构由开始、输入、方程求解、输出和结束等模块组成。由于不同情况输出信息不同，因此，算法输出被放在相关模块中。方程求解模块包含嵌套使用的两个选择结构，分别用来控制 a 是否等于 0 的两种不同执行路径的选择，以及是否可以开根号的两种不同执行路径的选择。

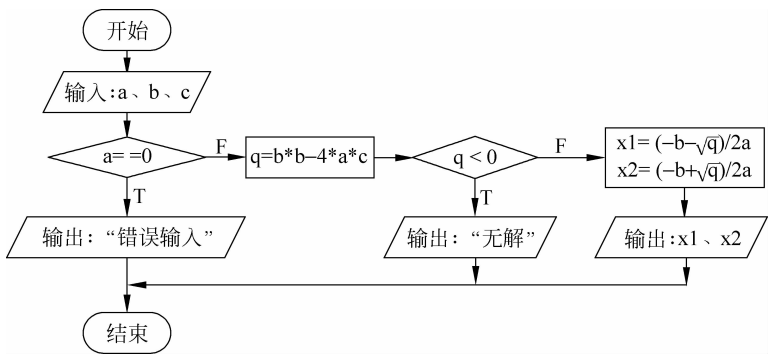


图　3.6

2）多分支选择结构设计

双分支选择结构比较简单，适用于描述不太复杂的分支现象，而多分支结构更具有普遍性，适用于描述更复杂的分支现象。

【例 3.6】 设计一个根据输入的月份序号输出相应月份英文名称的算法。

问题分析：本问题的要求是：首先，输入一个代表月份的整数，这个整数的取值范围是 1～12；其次，根据输入的整数确定对应的英文月份名；最后，输出确定的英文月份名。该问题的数学模型是：

$$f(Month)=\begin{cases}\text{“January”} & Month=1\\ \text{“February”} & Month=2\\ \text{“March”} & Month=3\\ \text{“April”} & Month=4\\ \text{“May”} & Month=5\\ \text{“June”} & Month=6\\ \text{“July”} & Month=7\\ \text{“August”} & Month=8\\ \text{“September”} & Month=9\\ \text{“October”} & Month=10\\ \text{“November”} & Month=11\\ \text{“December”} & Month=12\end{cases}$$

数据结构：Month 为整型变量，存储输入的月份值，“January”，“February”，…，“December”为 12 个月份的英文名（字符串常量）。

算法流程图：根据本题的数学模型，可以看出 f(Month)的值取决于 Month 变量的值。显然，可以使用多分支选择结构，其中的判断条件是输入的代表月份的数字，根据这个数字的具体取值，输出相应的英文月份名称，具体描述如图 3.7 所示。

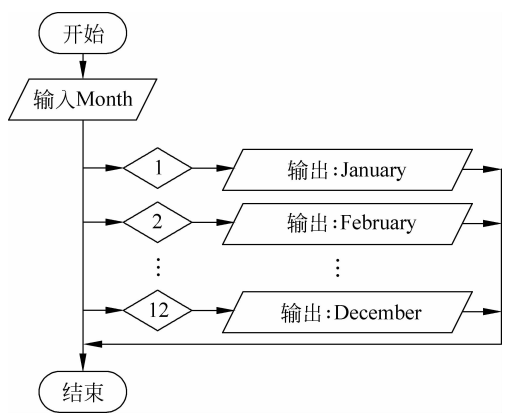


图 3.7

本题也可以使用嵌套的双分支结构，但是，使用多分支结构描述本题的算法条理更加清晰，层次更加分明，能够提高算法的可读性。

4. 循环结构设计

循环现象在客观世界普遍存在，例如星体运动、日月更替、现代化生产线的运行等，这些事物的发展过程或运动过程总是有规律地重复一些相同或相似的环节。与此类似，在算法设计中把那些需要反复执行某个模块的现象称为循环，把被重复执行的模块称为循环体。

循环结构是组成结构化算法的 3 种基本结构之一，这种结构由循环条件和循环体构成，通常分为当型（while）和直到型（until）两种，这两种循环的区别在于当型循环的循环体将被执行 0 次或多次，而直到型循环的循环体至少被执行一次。为与人们熟悉的 for 形式循环

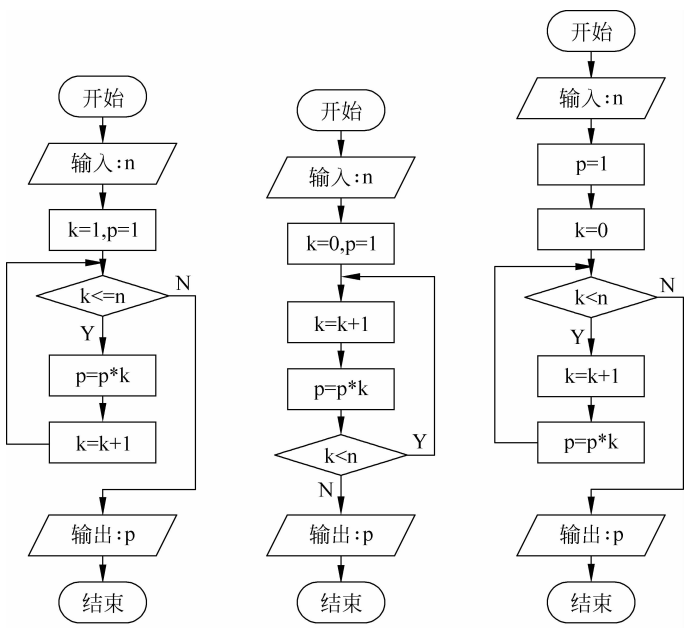
相对应，引入步长型循环，即每执行一次循环体，被指定作循环变量的变量值增加一个步长，但这种循环类型可以看作是前面两种循环的特例。当型循环如图 3.8(a)所示，直到型循环如图 3.8(b)所示，步长型循环如图 3.8(c)所示。

【例 3.7】 求任意正整数的阶乘。

问题分析：本问题的要求是：首先，输入一个正整数；其次，计算输入的正整数的阶乘；最后，输出计算结果。该问题的数学模型是：n!＝1×2×…×(n－1)×n。

数据结构：n，k，p 均为整型变量，分别存储待求阶乘的正整数、已循环次数和累乘积。

算法流程图：根据该问题的数学模型可知，求 n 的阶乘实际是不断地循环做乘法，每次的两个乘数分别为上次乘法的乘积以及目前已经循环的次数，直到循环 n 次为止。因此，设计的 3 种算法流程图如图 3.8 所示。



(a) 当型循环　(b) 直到型循环　(c) 步长型循环

图　3.8

3.3　算法的计算复杂性

解决一个具体问题的算法通常有多种，要知道哪一种算法比较好，需要对算法执行效率进行分析，包括时间效率和空间效率，分别被称为时间复杂性(time complexity)和空间复杂性(space complexity)。

时间复杂性描述算法执行时占用计算机时间资源的情况，是一种抽象的描述方式，不是指与算法实现效率有关的算法执行时间，而是指理论上与问题规模、算法输入及算法本身相关的某些操作次数的总和，通常记为 T(n)。问题规模逐渐增大后时间复杂度的极限形式称为渐进时间复杂性(asymptotic time complexity)，渐进时间复杂性确定算法所能解决问题

的规模，通常被用来分析随着问题规模的加大，算法对时间需求的增长速度。

算法的空间复杂性是指算法需要消耗的空间资源。其计算和表示方法与时间复杂性类似，一般都用复杂性的渐近性表示。与时间复杂性相比，空间复杂性的分析要简单得多。

3.4 常用算法设计策略简介

设计算法时应根据具体问题采用适当的技术手段和策略，下面简要介绍算法设计中经常使用的几种策略和技术，更详细更深入的知识可参阅有关文献。

1. 分治法

分治法就是将问题分而治之，是人们解决复杂问题的经常选择。其思想是：将一个复杂问题分解成一系列简单的、易解决的组成部分，逐一将所有组成部分解决后，再将它们组合到一起，得到复杂问题的解答。分治法通常与递归技术一同使用，是一种解决复杂问题的有效策略。

2. 递归技术

算法与程序设计中的递归技术是一种狭义的递归，是指将问题划分成不同层次的子问题，解决每一层问题的难度随着层次的降低而减小，解决高层次问题的充分条件是其低层次问题得以解决，这些不同层次的问题具有极大的相似性，以至于解决所有这些问题的算法相同（只是输入不同），这样就可以在解决某层次问题的算法中调用同一算法解决低层次的问题，这种嵌套的算法调用在运行时反复调用相同的算法解决比其低一层次的问题，直到最低层次问题的解决为止，然后将逐级回代直至所有高层次问题得到解决。

3. 贪心法与回溯法

贪心法是一种求取最好/优结果的算法，它在每一步选择中都选取所处状态下最好/优的选择，从而希望导致结果是最好/优的。贪心法可以解决一些最优性问题，如求图中的最小生成树，求哈夫曼编码等。对于其他问题，贪心法一般不能得到所要求的答案。如果一个问题可以应用贪心法解决，那么贪心法一般是解决这个问题的最好办法。由于贪心法的高效性以及所求得的答案比较接近最优结果，贪心法也可以用作辅助算法，或者直接解决一些要求结果不特别精确的问题。

回溯法是一种选优搜索法。回溯法按选优条件向前搜索，以期达到目标，但是，当搜索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术被称为回溯法，满足回溯条件的某个状态点被称为“回溯点”。

用回溯法解题的一般步骤：

（1）针对要求解的问题，定义问题的解空间；

（2）确定易于搜索的解空间结构；

（3）以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。