Section 1.3 Winning Solutions 竞赛中的策略

一个获得优势的好办法是写下你做题的策略。这会使你的思路很清晰，很清楚你做的东西是对是错。这样你是在先用你的思考时间来找出你的错误，而不是直接去想下一步应该怎么做……也就是先想好了再做。

心理准备也很重要。

竞赛中的策略

首先通读题目，然后写出它的算法、复杂度、数据规模、数据结构、程序细节……

- 想想所有可能的算法——然后选有效的中最笨的！
- 做数学计算！（时空复杂度，最坏的和期望的）
- 试着打破算法——利用特殊（让算法退化？）的测试数据（感觉是条件，但 test cases 就是测试数据）
- 做题的顺序：先做最简短的，根据你的情况（顺序（用时从短到长）：做过的、简单的、不常见的、难的）

编写程序代码——每个程序一次解决：

- 定下算法
- 想出特殊的测试数据（最狡猾的）
- 写出数据结构
- 写 input 代码，调试（写额外的输出程序去检测，笔者猜是“对拍”吧？）（后译者注：原句 write extra output routines to show data？个人认为意思是写额外的输出程序显示输入数据，估计目的为了确认输入处理正确——这个绝对有惨痛教训的。）（另一译者：另外一定要注意完成最终代码的时候删除这些输出语句！）
- 写 output 代码，调试
- 逐步完善：写注释，写出程序的思路
- 写出代码，一段一调试
- 运行&查正确性（使用特殊的数据）
- 试着去 break 代码的正确性——使用特殊的测试数据
- 不断优化——根据需求（使用极端的测试数据来测试运行时间）

时间控制 和 降低损失 scenarios

当出现错误的时候，有一个调试的计划；想想程序是什么样的，你希望它有什么样的输出？重点问题是：“什么时候你要去查错，什么时候你要放弃去做下一道题？”思考这些问题：

- 你已经用了多长时间去查它的错？
- 看起来这是什么类型的错误？
- 是你的算法错误吗？
- 是不是你的数据结构需要变化？
- 你有没有一些线索，错误在哪？
- 短时间的查错（20 分钟）比转去做别的题好；但是你可能用 45 分钟解决另一道题。
- 什么时候去回头看一些你已经放弃的问题？
- 当你已经花了很多时间去优化一道题，什么时候应该去看下一道题？
- 想到这——忘记之前，想想，从现在开始：怎样你才能在接下来的时间里得到最高的分数？

有一个 checklist，在交上你的代码之前：

- 在竞赛结束前 5 分钟停止对代码的修改？
- 停止维护。
- 停止调试性的输出。

技巧&诀窍

- 穷举，如果能这么做
- 保持简单，傻瓜 (KISS = Keep It Simple, Stupid)：简化就是聪明！

- 要点：注意限制条件（题目描述）
- 浪费内存空间吧，当它会让你的生活变得容易时。
- 不要删除你调试的额外输出，注释掉它。
- 不断优化，但是只要满足你的需求就可以了。
- 保留下所有的代码版本！
- 调试代码：
  - 空格是个好东西
  - 使用有实际意义的变量名
  - 不要重复使用变量
  - 逐步完善
  - 在代码之前写注释
- 可以的话，尽量避免指针
- 像避免灾难一样避免动态分配内存：静态分配所有变量。
- 试着不要去用浮点数 ；如果不得不用，在所有的地方去容差（不要用 == 判断相等）
- 对注释的评论：
  - 不要大段散文，只要简单的注释。
  - 解释高级的功能：++i; /* 把 i 自增 */ 写出这样的注释比没有任何注释还糟糕
  - 解释难懂的代码
  - 分割&功能的模块化
  - 让聪明的人懂你的程序，而不是代码
  - 所有的事情你都要去思考
  - 对所有你第一次看到的东西，说“我怎么把它再做一遍？”
  - 总是去说明每个数组的意义
- 记录你每一次比赛的表现：优点、错误、哪些地方可以做得更好；用这些来重写一遍，改进你的比赛计划！

复杂度

基础和命令符号 略

经验

- 当分析对于一个给定的数据，需要运行多长时间，首先一个常识是：现在（2004）计算机 1s 可以处理 100M 的内容。在一个时限 5s 的程序中，大概可以有 500M 的动作。好的优化可以让这个数字×2 甚至×4。算法的复杂度大概只能到这个数的一半。现在的竞赛常常对很大的数据给出 1s 的时限。
- 最多用 16M 内存
- 2 的 10 次方≈10 的 3 次方
- 如果你在 N 次迭代中，每次有 k 层循环，那你的程序有 O(N 的 k 次方)的复杂度。
- 如果你有 L level，每 level 有 b 层递归调用，那么复杂度是 O(b 的 L 次方)。
- 记住，N！是排列，2 的 n 次方 是 子集 或 组合。
- 对 N 的排序最好的时间复杂度是 O(N log N)。
- **做数学计算！**Plug in the numbers.

算复杂度例子： 略

解决问题例子

直接生成 VS 爆搜

（原文：Generating vs. Filtering，直译：生成 VS 筛选，怪怪的）

计算出非常多可能的解然后选择一个正确的（比如八皇后问题）方法是爆搜，一开始就计算可行解的方法是直接生成。一般，爆搜比较容易写（写得也快）但运行得慢。估算以确定题目规模允许爆搜还是不得不需要找出一定的算法。

预运算

有的时候打表或用其他数据结构能够使结果可能更快地被找出。这叫做预运算（也可以说是用空间换时间）。你可以把计算好的结果写到程序中编译，也可以在程序开始运行时计算，也可以让程序记下之前运算的结果。比如说，一个必须把大写字母转换为小写字母的程序可以打出一张对应表。竞赛中经常需要用到素数——许多时候比较常见的方法是先运算一张素数表然后在其他地方调用。

分解（在竞赛中最难做到）

虽然有少于 20 个基本算法在竞赛问题中用得到，但是解决由两个算法组合的问题是令人望而生畏的。试着把问题分割，让您可以结合循环或另一种算法来独立解决问题的不同部分。请注意，有时你可以在数据不同部分使用相同的算法两次（独立的!）来显著降低您的运行时间。

对称性

许多问题都有对称性（比如说，一对点间的距离从两种方向遍历是相同的）。对称性可以有两路、四路、八路甚至更多。试着利用对称性来降低程序运行时间。

举例来说，利用四路对称，你只需解决四分之一的问题然后利用对称性写出其余答案。（当然，要注意一些自对称方案只需输出 1 次或 2 次）

从前往后还是从后往前

令人惊奇的是，对于许多竞赛问题的测试数据中从后往前比从前往后运算要好很多。注意逆序处理数据或者构造一些常规数据以外的特别的顺序或题目中流行的测试数据。

简化

有些问题是可以改写成一个有点不同的问题，让你认为就像解决新问题，你可能对原来的问题有或很容易想到解决方案。当然，你应该解决是两个中容易的。另外，对一些问题可以用归纳法，先做点改变解决一个小问题然后找到完整解法。