论程序的调试技巧

【关键字】调试技巧、测试方法、测试用例设计

【摘要】本文结合作者自身经验，对竞赛中程序的调试技巧做了详细的阐述和总结。在介绍了编程中常见的错误类型和集成环境的调试工具之后，给出了一般调试流程，并着重讲述了其中的动态查错技巧，做了一定的归纳。最后通过一个调试实例来体现本文所论述的调试技巧的具体应用。

【正文】

1、 程序调试的必要性

程序设计过程中，错误是在所难免的。虽然有些程序员认为一个程序可以做到完美无瑕，但实际情况却并非如此，不然就不会有人对 Windows 怨气冲天了。尽管信息学竞赛中所编的程序从来不会像 Windows 那样庞大，最多也是仅仅几百 K 而已，但由于时间有限，选手们的程序难免有疏漏之处。因此，调试就成了极其重要的一环。如何在紧迫的时间内快速准确地发现并改正错误，正是本文所要讨论的问题。

2、 常见错误类型归纳

《孙子兵法》云：“知己知彼，百战不殆。”对于程序调试者来说，程序中的错误就好比是敌人，如能准确把握敌人的情况，无疑是极为有利的。下面我们就来对常见的一些错误类型进行归纳并给出解决方法。

1、 思路错误

这要看是基本算法错误还是功能缺陷。前者需要重写大部分代码，是否重写则根据时间是否充裕而定，后者只需增加一部分代码，再修改某些地方，这时应全面考虑，以防遗漏应该修改的地方。

2、 语法错误

这个没什么可说的，作为一名信息学竞赛的选手，应该对自己选择的编程语言的语法了如指掌，具体在这里就不多讲了。

3、 书写错误

这种错误令人十分头痛，一般的书写错误在编译时都能找出来，但如果你在表达式中用到变量 j 时误写成了 i，不但编译程序找不出来，自己找时也由于两者样子比较相似，难以发现。排除这种错误只能靠“细心”两字，具体可使用下面要介绍的静态查错法。

4、 输出格式错误

由于现在信息学竞赛采用黑箱测试法，由于输出格式错误而导致失分的例子屡见不鲜。一个标点，一个空格，都会导致最后的悔恨。因此，在调试时先要

核对输出格式，针对不同输出格式多设计几个测试用例，以防一失足成千古恨。

5、 其它编程时易犯的错误

除了上面所说的错误类型外，其它就属于编程时在细节上考虑不周所造成的了。下面仅列举其中一些较为隐蔽的错误。只有靠平时不断总结积累，才能真正的做到“知己知彼”。

①变量未赋初值

看下面的程序段

```
For i：=1 to N Do
 If A[i]>Max Then Max：=A[i]；
WriteLn（Max）；
```

这个程序段的原意显然是要输出数组 A 中最大的数。但由于它遗漏了将 Max 赋初值的语句，因此很可能会出现输出的数并不在数组 A 中的错误。应该在过程开头添上一句 Max：=-MaxInt；。养成变量使用前先赋初值的习惯能预防许多较隐蔽的错误。

2　中间运算越界

看下面这两句语句

```
A：=1000；
B：=A*A Div 100；
```

其中 A，B 都是 Integer 类型。按照我们的想法，1000*1000 Div 100=10000。然而当我们察看 B 的值的时候，却发现 B 等于 169。原因是 Pascal 在进行编译时总是先计算出 A*A，把它放到一个中间变量中，然后再计算出最后结果放入 B 中。而 A*A 超出了 Integer 的范围，这就是造成错误的根本原因。要使 Pascal 能报告这类错误，只要打开编译开关 Q 即可。对此类错误解决方法是使用强制类型转换，写成 B：=LongInt（A）*A Div 100。编译程序会自动把中间变量规定为 LongInt 类型，就不会越界了。

3　局部变量与全局变量同名造成概念混乱

这个实际上不能算错误，然而有许多错误正是因此而起。一个常见的错误是当我们在过程中使用全局变量时，忘记了自己在该过程中还定义了一个同名的局部变量，从而使得实际的程序与我们的思路不一致；另一个常见的错误是局部变量忘记定义，在过程（函数）中实际上使用的是全局变量，而出现错误往往是在这个过程（函数）之外某个需要使用该全局变量的地方，这就增加了调试的难度。因此，应该尽量避免使用同名变量。

4　If-Then-Else 语句混乱

Pascal 对 If-Then-Else 语句的规定是：If-Then 语句可以没有 Else 语句与之相匹配；Else 语句总是匹配最近的 If-Then 语句。这一点使得我们在使用嵌套的 If-Else 语句时容易出错。如下面这个例子：

```
If 条件 a
 Then If 条件 b Then 代码段 b
 Else 代码段 a
```

我们的原意是让代码段 a 在条件 a 不成立时执行，但由于 Else 语句总是匹配最近的 If-Then 语句，因此这个 Else 是与 If 条件 b Then 这个语句相匹配的，也就是说代码段 a 要满足条件 a 成立且条件 b 不成立时才会执行，与我们原意相去甚远。解决方法是在需要的地方加一个空的 Else，就如上面的例子，要在 If 条件 b Then 语句后面加一个空的 Else。

5　实数比较出错

在比较两个实数是否相等时，如果直接用等号，往往会造成错误。这是浮点运算存在误差所造成的，解决办法是使用两数差的绝对值与一个相对极小量进行比较，一般说来如果 abs(a-b)<1e-8，则可认为 a=b。

3、集成环境的调试工具

对于一个战士来说，对自己手中的武器性能特点应该了如指掌。对于程序调试者来说，调试工具就相当于武器，熟练掌握调试工具，充分发挥它的性能，对于迅速找出错误，加快我们的调试速度有着极大的帮助。下面就对集成环境提供的调试工具做一些介绍。

调试时主要使用的两个菜单是 Run 和 Debug。Run 菜单提供了各种程序执行方式，而 Debug 菜单提供了对变量的观察，修改以及断点等功能。

程序的执行方式有四种：

1、 Run，运行整个程序（Ctrl+F9），该方式常用在总体测试上。一般每一个测试用例都应先用该方式执行程序，如果输出答案正确就可以直接转到下一个测试用例，免去了不必要的时间。即使发现错误也只不过比直接进入模块调试增加了一点点时间，是完全值得的。

2、 Step over，单步执行，把整个过程（函数）视为单步一次执行（F8），该方式常用在模块调试时期，可以通过观察变量在模块执行前后的变化情况来确定该模块中是否存在错误，也可以用来跳过已测试完毕的模块。

3、 Trace into，单步执行，对于过程（函数）进入到内部一步步执行（F7），该方式常用在底层调试时期，可以跟踪程序的每步执行过程。它的优点是容易直接定位错误，缺点是调试速度较慢，尤其是当错误位于程序后部时。所以一般是采用先用模块调试法尽量缩小错误范围，然后使用第 4 种执行方式和断点来快速跳过没有出现错误的部分，最后才是用该方式来逐步跟踪找出错误。

4、 Go to cursor，执行到光标处（F4），之所以把这种方式放在最后介绍，是因为这种方式的灵活度较大，不但可以一次执行一行，也可以一次执行多行可以直接跳过过程（函数），也可以进入过程（函数）内部。它有断点的定位能力强的优点，又比断点更加灵活。正确适当地使用这种方式可以大大加快我们调试的速度，这需要靠丰富的调试经验。可以参考后面的调试实例。

Debug 菜单中最常用选项是 Watch 和 Add Watch，这两个用于跟踪观察变量和表达式在程序执行过程中值的变化，这样就可以随时检查它们是否按照算法要求输出，是否符合正确答案。大多数错误在调试时都可以只使用它们以及上面的四种执行方式被检查出来。

有的时候，虽然知道该模块有错，但一时无法找到错误所在，并且上面所讲的后三种执行方式都难以快速定位。例如对于一个程序，我们需要它执行到某个语句并满足某个条件时停下来，Go to cursor 只能保证执行到这个语句时停下来，却不能保证满足条件，这时我们便需要使用断点。断点虽然没有 Go to cursor 灵活，但有一个 Go to cursor 无法取代的优势便是它可以设置中断条件。使用 Add Breakpoint 选项（Ctrl+F8）可以在当前光标处设置一个断点，Breakpoints 选项可以编辑断点条件。要注意的是，断点的设置会大大降低程序执行效率，因此调试完毕以后一定要记得清除所有断点。

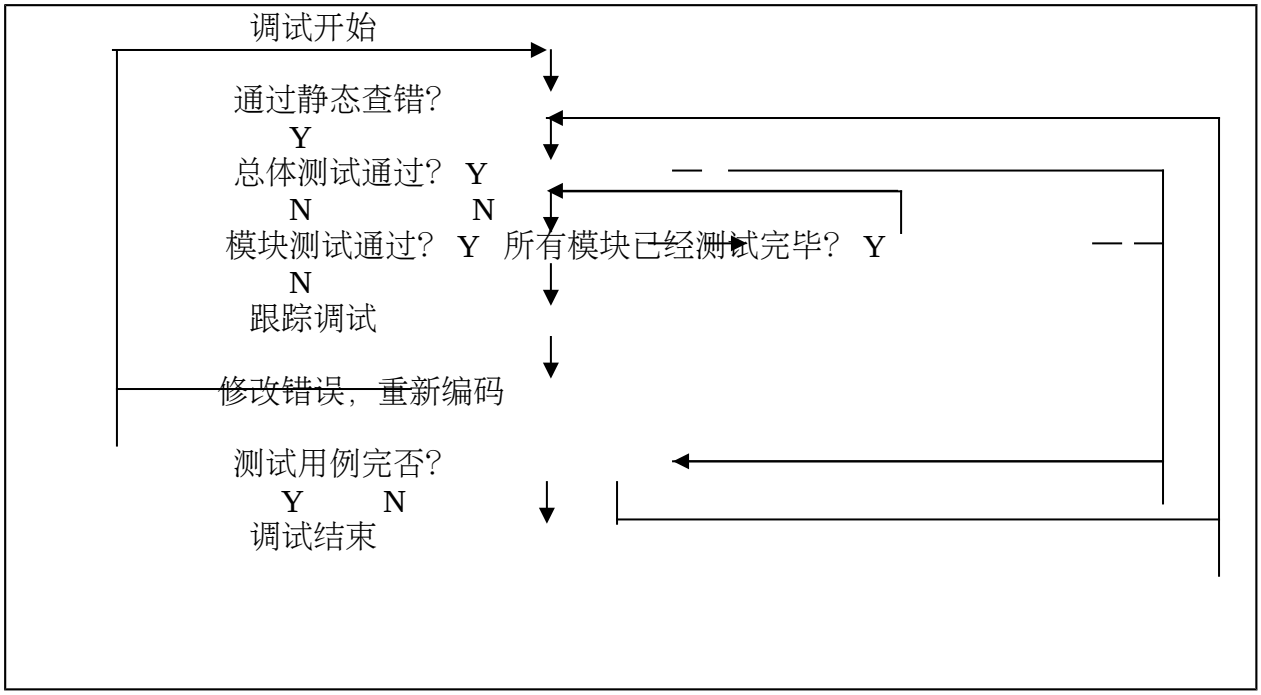
Evaluate/modify 选项（Ctrl+F4）用于临时观察修改表达式的值，如果在调

试过程中，需要临时观察或修改某个表达式的值，则可以打开 Evaluate and modify 窗口进行操作。这个方法尤其适用于对过程（函数）的调试，我们可以先用 Go to cursor 执行到某个过程（函数）的开头，然后使用 Evaluate/modify 选项改变参数的值用于检查不同情况下这个过程（函数）的执行结果。但是要注意，一个过程（函数）通常不是孤立的，它经常需要使用某些全局变量，因此仅改变参数而不改变其他全局变量的值有可能是非法的。所以需要特别的小心，避免出现这样的情况。

Call stack 选项（Ctrl+F3）用于显示当前堆栈状态，这特别适用于对递归算法的调试。我们可以利用它察看每层参数的传递情况。不过一般来说参数的传递不易出错，因此该选项的用处并不是很大。

4、 调试的一般过程

对于一道题我们一般按照以下流程图进行调试:



这只是针对一般情况而言，在具体调试时，可以改变某些步骤，比如说在发现某个模块有错误改正以后，可以不返回到静态查错阶段，而是继续该模块的查错。这要视具体情况而定。

下面我们对各个步骤作详细的叙述。

1、 静态查错

静态查错是指不执行程序，仅根据程序和框图对求解过程的描述来发现错误。由于有些错误运行时很难查出，但在静态查错中却容易发现，比如说前面说到的书写错误。因此，静态查错往往是不可忽视的审查步骤。静态查错一般顺序为先查程序局部，后查程序整体。查局部主要是独立地检查各个子模块的功能是否按照算法要求实现，查整体主要是检查各个局部之间的接口是否吻合，是否缺少某些功能。在静态查错阶段，我们可以有针对性地检查上面所列举的错误类型。在这个阶段查出的错误越多，节省的调试时间也就越多。

2、 动态查错

静态查错能够查出错误，但无法保证没有错误，因为这里有一个人为的因素在里面，只要一不小心，就可能漏掉一个错误。因此，我们需要动态查错与之相结合来找出遗漏的错误。与静态查错相对地，动态查错是指通过跟踪程序的执行过程，核对输出结果来发现错误。动态查错的技巧可分为两大部分：测试用例的设计和测试的方法。我们先来讲测试方法，

动态查错的测试方法可以按照两种标准进行分类。

1 按照测试用例的设计依据的不同，可分为黑箱测试法和白箱测试法。

只知程序的输入与输出功能，而不知程序的具体结构，通过列举各种不同的可能情况来设计测试用例，通过执行测试用例来发现错误的测试方法叫做黑箱测试法。已知程序的内部结构和流向，根据程序内部逻辑来设计测试用例，通过执行测试用例来发现错误的测试方法叫做白箱测试法。在竞赛中我们常常使用两者结合的方法进行测试。在进行底层模块测试的时候可以使用白箱测试法，通过专门的测试条件和测试数据来考虑程序在不同点上的状态是否符合预期的要求。在总体调试的时候则可以使用黑箱测试法，脱离程序内部结构来考察对于不同情况下的测试数据程序是否能够正确出解。对于中间模块，可以用黑箱，也可以用白箱，或是两者兼用，具体要看适应那种测试法。一般说来，结构复杂的模块使用黑箱测试法，结构简单的使用白箱测试法。最后要说的是，由于白箱测试法测试用例设计比较困难，并且现在信息学竞赛都采用黑箱测试法进行测试，所以我们在竞赛时间紧张的情况下，可以一律采用黑箱测试法，这样效率比较高。

2 按照测试顺序的不同，可分为由底向上测试和从顶向下测试。

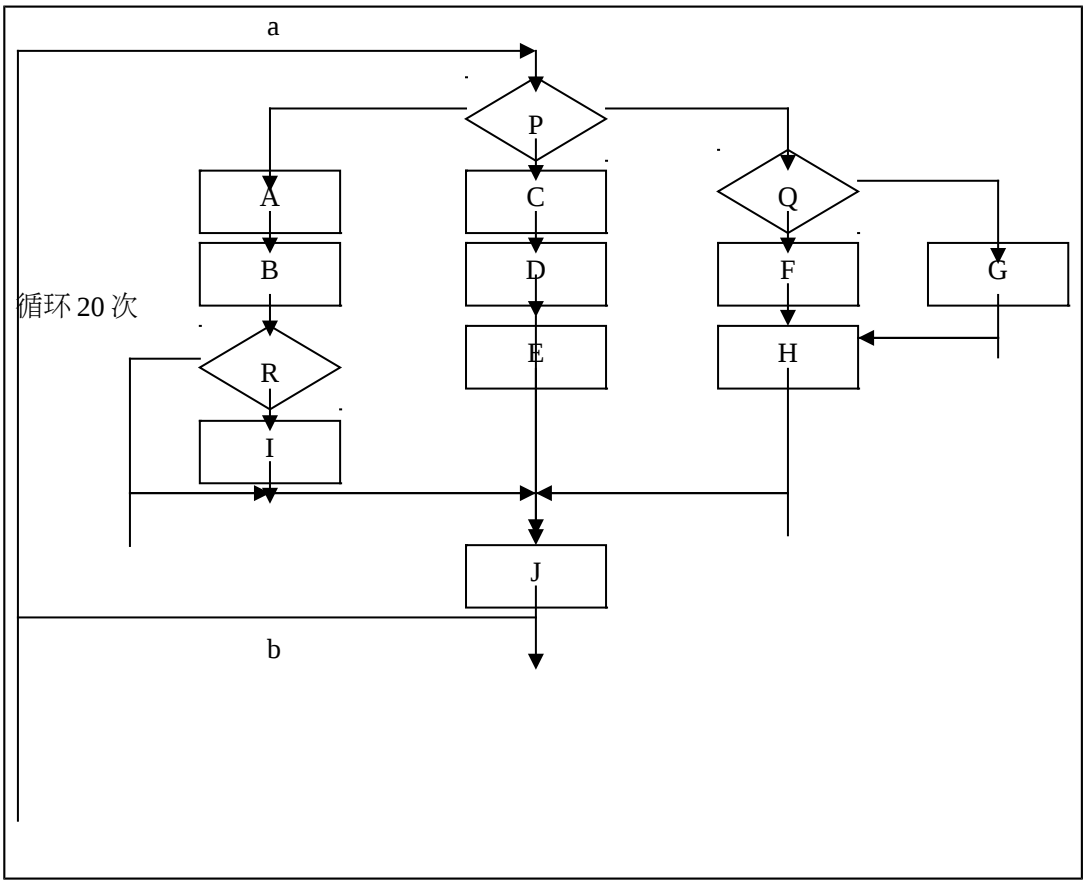
由底向上测试是先测最底层的模块，然后依次向上测试，最后测试主模块。从顶向下测试刚好与之相反，先测主模块，然后按照从顶向下设计的顺序依次测试各个模块。为了加快调试的速度，建议采用从顶向下的测试顺序，只在发现某个模块有错时才进入下一层调试，这样对迅速定位错误也有很大帮助。

在运用这些测试方法时，我们要注意哪些问题呢？首先，对自己所编的程序的结构、模块的功能一定要了如指掌。采用从顶向下的测试方法时，经常是一个模块还没有测试完，就转到了下一个模块，因而特别容易忘记和疏漏。如果对程序结构心中没有概念，就很容易被弄糊涂。又如果对模块的功能不是很清楚，则难以判断模块执行结果的对错，从而无法准确确定错误所在。其次，测试需要有条理地进行。坚持使用同一个测试用例直到输出正确为止；在一个模块没有测试完毕时，不要进行下一个模块的测试，除非这个模块是当前模块的子模块且在当前模块的测试中发现这个子模块有错。最后，在每次修改了源代码之后一定要把已经测过的所有测试用例再测一遍，以防产生新的错误。

讲完了测试方法，我们再来讲讲测试用例的设计。

黑箱测试法的测试用例是根据输入数据的不同情况来设计的。由于一道题的输入数据可以千变万化，因此黑箱测试法不可能测遍所有的情况。如有这样一道题，已知程序要求输入两个 LongInt 类型的整数 X、Y，输出的是它们的和。X 共有 2^32 个不同值，Y 也有 2^32 个不同值，这样不同的输入数据共有 2^32*2^32=2^64 种不同情况。假设执行一遍程序要 1 毫秒，那么要测遍所有的不同情况大约需要五亿年，显然是不可能做到的。

白箱测试法的测试用例是根据程序内部逻辑来设计的。要完全测试整个程序，测试用例就必须覆盖程序的所有执行路径，这通常也是不可能的。例如，有一程序的流程图如下：



其中从 a 到 b 有五条路径，再外套循环 20 次，根据乘法原理，执行一遍程序可能经过的不同路径共有 5^20≈10^14 条 。假如程序执行一遍从 a 到 b 要 0.1 秒那么测试完所有路径就需要一百多万年，这显然也是不可能的。

由上面两例可以看出，动态查错和静态查错一样，只能发现某些错误的存在，而不能证明错误的不存在 。所以，我们在设计测试用例的时候，需要考虑测试用例的经济性。

所谓经济性是指用尽可能少的测试用例来覆盖尽可能多的情况，对于黑箱测试法来说，就是要包括尽可能多的不同的输入数据类型，对于白箱测试法来说，就是要覆盖尽可能多的执行路径 。考虑到在竞赛中的测试以黑箱测试法为主我们仅讨论黑箱测试法的用例设计。常用的设计方法有等价分类法、边界分析法和错误推测法等等。

①等价分类法

所谓等价分类法就是根据程序功能将输入的数据划分成若干个等价类，然后考虑数据选择，设计出测试用例，以达到测试目的。

有这样一道题：输入三个整数表示一个三角形的三条边长 。要求输出能否构成一个三角形，如能则输出是等腰，等边还是既非等边又非等腰三角形。

对于这道题，我们运用等价分类法，根据三角形是否合理可以将输入数据分成两个等价类：合理三角形和不合理三角形 。对于合理三角形，我们可以继续将该等价类分为等腰三角形和既非等边又非等腰三角形，而对于等腰三角形，还可以分为单纯的等腰三角形和等边三角形 。这样我们通过不断地等价分类最后共可以设计出四种测试用例。另外还可以根据输入数据的不同排列方式来分类。但如果你的程序是先排序再进行判断，而且能够保证排序正确性，就没必要使

用这种分类方法。

②边界分析法

程序运行出错常常发生在边界状态，所以在测试中我们常首先根据程序的功能确定边界情况的数据变化，以便设计测试用例。这种对边界状态进行分析，以设计测试用例来测试程序的方法称为边界分析法。

边界值的选择要根据题目实际情况有针对性地，有一定创造性地进行。一般来说，可考虑如下几种情况：

1 恰好在边界附近，且欲越界的数据；
2 取最大或最小值，最多或最少值加减 1 的数据；
3 循环或迭代的初始值与终值数据；
4 有序集合的第一个或最后一个数据元素；
5 零点附近的元素；
6 最大误差值的数据；
7 数据量最大或最小的数据
8 计算量最大或最小的数据

仍然使用上面那道三角形的题目作为例子，有如下三个测试数据：

a、两边之和等于第三边；
b、 三个数中包含 0；
c、三个数均为 0；

a 属于第㈠种情况，b、c 属于第㈤种情况，这就是边界值分析法的应用。

边界分析法是很重要的一种方法，是一般测试中必不可少的。它比较精炼，又容易发现错误。问题在于有些程序的边界情况是极其复杂的，有时甚至不存在比如说让你把 10 个数从大到小排序输出，因为算法只与两个数之间的大小关系有关，与每个数的数值大小无关，数的总数也是固定的，这时就不存在边界情况，也就无法使用边界分析法来设计测试用例。

③错误推测法

错误推测法实际上是利用了黑箱白箱结合的思想，根据自己设计的程序来找出容易出现错误的地方，然后有针对性地设计测试用例。例如在一个有许多 If-Then-Else 语句嵌套的地方，就很容易出现错误，而一般的测试用例很难找出这种错误。这时错误推测法就能派上大用场，对于这一系列的 If-Else 语句，设计出覆盖不同路径的测试用例，从而检验程序的正确性。

至于具体测试时选用哪种方法，我们常用的策略是首先用边界分析法，再用等价分类法加以补充。最后对于程序中结构复杂的部分重点使用错误推测法进行测试。当然在时间允许的情况下，应该使用更多的测试数据来覆盖更多的功能

3、 跟踪调试

跟踪调试通常是一件繁琐的事。它需要选手的耐心和细心。选择哪些变量进行跟踪也是至关重要的，准确的变量选择可以起到事半功倍的效果。一般来说，首先要跟踪的是存放输入数据的变量，尤其对于那些需要对输入数据进行一定处理的题目来说更是如此，输入数据不正确，即使是正确的程序也会与答案不符合；其次是那些频繁用到的全局变量，这些变量往往贯穿于整个程序中，一旦某处出错，会影响到其他模块的正确性，由此造成定位错误。出错的地方没有测试，正确的地方反而反复测试。因此对于这些全局变量的变化要密切加以关注不可放过任何错误；再次就是那些循环变量了，跟踪循环变量可以准确地得知程序的执行进度，从时间上把握错误所在；最后其他的变量则要根据实际情况

加以选择，对使用较多的变量应优先加以跟踪。

另外 Watch 窗口的大小位置也要合理安排，使得在跟踪过程中重要的变量的变化情况能够一目了然，从而免去许多不必要的窗口切换，节省时间。

5、 总结

在信息学竞赛中，程序效率高低并不是首要的。不正确的程序效率再高，也是等于零。效率低的程序只要正确，总是能拿到一点分数。因此调试水平的高低很大程度上也反映了一个选手整体水平的高低。总的来说，程序的调试主要靠的还是经验。经验越丰富，调试效果也就越好。因此我们平时训练时决不能只注意对思路算法的训练，还应该注意训练自己的调试水平。

6、 一个调试实例

题目：

在一些美国主要城市里，为企业传送文件和小物品的自行车快递长期以来就是流动运输服务的一部分。波士顿的骑车人是不同寻常的一族。他们以超速、不遵守单行道和红绿灯、无视汽车、出租、公交和行人的存在而臭名远扬。快递服务竞争激烈。比利快递服务公司（BBMs）也不例外。为发展业务，制定合理的收费 BBMS 正根据快递员能走的最短路线制定一项快递收费标准。而你则要替 BBMS 编写一个程序来确定这些路线的长度。

以下假设可以帮助你简化工作：

●快递员可以在地面上除建筑物内部以外的任何地方骑车。

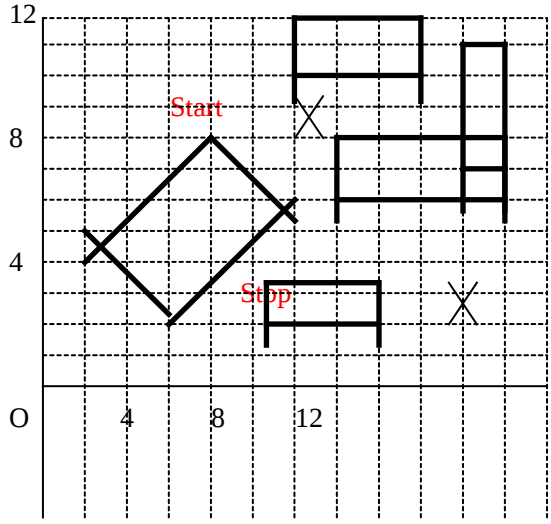
●地形不规则的建筑物可以认为是若干矩形的合并。并规定，任何相交矩形拥有共同内部，而且是同一建筑物的一部分。

●尽管两个不同的建筑物可能非常接近，但永远不会重叠。（快递员可以从任意两个建筑物之间穿过。他们能够绕过最急的拐弯，可以贴着建筑物的边缘疾驶。）

●起点和终点不会落在建筑物内部。

●总有一条连接起讫点的线路。

下图是一张典型的鸟瞰图。



输入文件包含如下若干行：

第一行：n

场景中描述建筑物的矩形个数。o<=n<=20

第二行：x1、y1、x2、y2

路线起讫点的 x 和 y 坐标。

余下 n 行：x1、y1、x2、y2、x3、y3

矩形三个顶点的坐标

所有 x 、y 坐标是属于 0 到 1000（包含 0 和 1000）的实数，每行中相邻的坐标由一个或多个空格分开。正如下面输出范例给出的那样。输出应给出从起点到终点的最短线路的长度，精确到小数点后第二位。

输入范例

5

6.5 9 10 3

1 5 3 3 6 6

5.25 2 8 2 8 3.5

6 10 6 12 9 12

7 6 11 6 11 8

10 7 11 7 11 11

输出范例

7.28

算法分析：

由输入文件的格式可以看出，每一个矩形输入三个顶点 P1，P2 和 P3。这三个顶点的坐标分别为（P1.x P1.y）、（P2.x，P2.y)、（P3.x，P3.y）。由于矩形的四条边不一定平行 x 轴和 y 轴，因此我们预先对 P1、P2、P3 进行排序，使得 P2 在 P1 的右方或者正上方，即（P2.x>P1.x） or （P2.x = P1.x） and （P2.y>=P1.y））；P3 在 P1 的右方或者正上方，即（P3，x>P1.x）or（（P3.x＝P1.x）and（P3.y>=P1.y））；P3 在 P2 的右方或者正上方，即（P3．x＞P2.x）or（（P3.x=P2.x）and（P3.y＞=P2.y））。然后求出矩形的第四个顶点 P4：

P4.x＝P1.x+P3.x-P2.x

P4.y＝P1.y+P3.y-P2.y

例如输入矩形的三个顶点（3，3）、（1，5）、（6，6）。按上述方法排序后 P1=（1，5），P2=（3，3），P3=（6，6）

P4.x＝p1．x＋P3.x-P2.x）＝1＋6 一 3＝4

P4.y＝P1.x+P3.y 一 P2.y＝5 十 6 一 3＝8

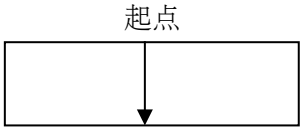
即 P4＝（4，8）

由排序的定义可以看出，对于每一个矩形的 4 个顶点来说，P1 和 P2 互为对角，P3 和 P4 互为对角。

由于快递员是在地面上除建筑物内部以外（包括建筑物边缘）的空间寻找一条最短路线，因此，这条路线上的顶点除起讫两个点外，其余都为矩形的顶点。最短路线上每条线段的两个端点既不能被任何建筑物所挡。也不能为同一矩形的对角点（不能穿过矩形对应的建筑物）。

我们将最短路线的起点、终点以及每个矩形顶点放入一个顶点序列表 P 中，其中 P[1]、P[4*n+2]存贮起讫点， P[2]…P[4*n＋1]存贮 n 个矩形的顶点。显然，

最短路线上的顶点取自于 P 表。在建立 P 表中的同时建立一张线段表 L，将每个矩形的四条矩形边和两条对角线存入该表。设置 L 表的目的是为了判断快递员的行车路线是否符合规则。如果行车路线与表中任一条矩形边相交，则说明快递员进入了建筑物内部，这种情况必须排除。问题是：为什么 L 表还要存贮每个矩形的对角线呢？这是为了剔除一种如下图的边界错误。



如果起点和终点贴在同一个矩形边上（这种情况是允许的，因为快递员可以贴着建筑物的边缘疾驶），则快递员将毫无顾忌地穿越“禁区”，因为两点间未相交任何矩形边。增设行车路线不能与矩形的对角线相交的条件，便可避免这个不易察觉的漏洞。

求最短路径则可采用标号法。

程序：

```
Program Corner;
Const
  FinName='corner.in';{输入文件名}
  FoutName='corner.out';{输出文件名}
  MaxBuilding=20;{最大建筑物}
  MaxPoint=82;{最多顶点数}
  MaxLine=120;{最多线段数}
  Zero=1e-8;{相对极小量，用于实数比较}
Type
  Location=Record{坐标类型}
    x,y:Real;
  End;
  Line=Array[1..2]Of Location;{线段类型}
Var
  Bld,{建筑物数}
  Pnt,{顶点数}
  Lne:Integer;{线段数}
  P:Array[1..MaxPoint]Of Location;{顶点序列}
  L:Array[1..MaxLine]Of Line;{线段序列}
  Dis:Array[1..MaxPoint]Of Real;{最短距离表}
Function GetMin(Var a,b:Real):Real;{返回两数较小值}
Begin
  If a>b Then GetMin:=b Else GetMin:=a;
End;
Function GetMax(Var a,b:Real):Real; {返回两数较小值}
Begin
  If a>b Then GetMax:=a Else GetMax:=b;
End;
```

```
Function GetMulti(p0,p1,p2:Location):Real;{计算叉积}
Begin
  GetMulti:=(p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
End;
Function Intersect(Var L1,L2:Line):Boolean;{判断两条线段是否相交}
Var
  M1,M2,M3,M4:Real;
Begin
  Intersect:=False;
  If (GetMin(L1[1].x,L1[2].x)<=GetMax(L2[1].x,L2[2].x))
    And(GetMax(L1[1].x,L1[2].x)>=GetMin(L2[1].x,L2[2].x))
    And(GetMin(L1[1].y,L1[2].y)<=GetMax(L2[1].y,L2[2].y))
       And(GetMax(L1[1].y,L1[2].y)>=GetMin(L2[1].y,L2[2].y)) Then
       {通过快速排斥试验}
    Begin
      M1:=GetMulti(L1[1],L2[1],L1[2]);M2:=GetMulti(L1[1],L1[2],L2[2]);
      M3:=GetMulti(L2[1],L1[1],L2[2]);M2:=GetMulti(L2[1],L2[2],L1[2]);
      If  (Abs(M1)>Zero)And(Abs(M2)>Zero)
      And(Abs(M3)>Zero)And(Abs(M4)>Zero)
      And(M1*M2>0)And(M3*M4>0) Then Intersect:=True;
    End;
End;
Procedure GetNextPoint(Var P1,P2,P3,P4:Location);
{将顶点排序并计算第四个顶点坐标}
Var
  T:Location;
Begin
  If (P2.x<P1.x)Or((P2.x=P1.x)And(P2.y<P1.y)) Then
    Begin T:=P1;P1:=P2;P2:=T;End;
  If (P3.x<P1.x)Or((P3.x=P1.x)And(P3.y<P1.y)) Then
    Begin T:=P1;P1:=P3;P3:=T;End;
  If (P3.x<P2.x)Or((P3.x=P2.x)And(P3.y<P2.y)) Then
    Begin T:=P3;P3:=P2;P2:=T;End;
  P4.x:=P1.x+P3.x-P2.x;
  P4.y:=P1.y+P3.y-P2.y;
End;
Procedure Init;{读入数据，并初始化}
Var
  i,j:Byte;
Begin
  ReadLn(Bld);
  Lne:=0;Pnt:=1;
  ReadLn(P[1].x,P[1].y,P[Bld*4+2].x,P[Bld*4+2].y);
  For i:=1 to Bld Do
```

```
    Begin
      For j:=1 to 3 Do
        Read(P[Pnt+j].x,P[Pnt+j].y);
      ReadLn;
      GetNextPoint(P[Pnt+1],P[Pnt+2],P[Pnt+3],P[Pnt+4]);
      For j:=1 to 3 Do
        Begin
          L[Lne+j,1]:=P[Pnt+j];
          L[Lne+j,2]:=P[Pnt+j+1];
        End;
      L[Lne+4,1]:=P[Pnt+4];L[Lne+4,2]:=P[Pnt+1];
      L[Lne+5,1]:=P[Pnt+1];L[Lne+5,2]:=P[Pnt+3];
      L[Lne+6,1]:=P[Pnt+2];L[Lne+6,2]:=P[Pnt+4];
      Inc(Pnt,4);Inc(Pnt,6);
    End;
  Inc(Pnt);
End;
Function GetDis(a,b:Byte):Real;{计算两点间距离}
Begin
  GetDis:=Sqrt((P[a].x-P[b].x)*(P[a].x-P[b].x)+(P[a].y-P[b].y)*(P[a].y-P[b].y));
End;
Function Visible(Var P1,P2:Location):Boolean;
{判断在 P1 点是否能看到 P2 点，即判断两点之间是否有建筑物阻挡}
Var
  TL:Line;
  i:Byte;
Begin
  TL[1]:=P1;TL[2]:=P2;
  For i:=1 to Lne Do
    If Intersect(L[i],TL) Then
      Begin Visible:=False;Exit;End;
  Visible:=True;
End;
Procedure Main;{主程序，用标号法求最短路径}
Var
  i,j,k:Byte;
  Min:Real;
  S:Set Of Byte;
Begin
  Dis[1]:=0;
  For i:=2 to Pnt Do
    If Visible(P[1],P[i]) Then Dis[i]:=GetDis(1,i)
      Else Dis[i]:=1e10;
  S:=[1];
```

```
  For i:=2 to Pnt Do
   Begin
    Min:=1e10;k:=0;
    For j:=2 to Pnt Do
     If (Not(j In S))And(Dis[j]<Min) Then Begin k:=j;Min:=Dis[j];End;
    If k=Pnt Then Break;
    S:=S+[k];
    For j:=2 to Pnt Do
     If (Not(j In S))And(Visible(P[k],P[j]))And(Dis[k]+GetDis(j,k)<Dis[j]) Then
      Dis[j]:=Dis[k]+GetDis(j,k);
   End;
  WriteLn(Dis[Pnt]:0:2);{输出答案}
End;
Begin
  Assign(Input,FinName);
  Reset(Input);
  Assign(Output,FoutName);
  Rewrite(Output);
  Init;
  Main;
  Close(Input);
  Close(Output);
End.
```

这个程序编译已经通过，现在我们进行静态查错。首先确定了函数 GetMin，GetMax，GetMulti 是正确的。因为这三个函数很短，也很简单，所以只要通过静态查错应该就可以了。接着往下看，发现在过程 Init 中有一行是 Inc（Pnt，4）；Inc（Pnt，6）；明显不对。程序的本意显然是要增加 Pnt 和 Lne 的值，所以应该把 Inc（Pnt，6）换成 Inc（Lne，6）。再往下，又发现在函数 Intersect 中计算叉积时连续给 M2 赋了两次值，这样前面一次赋值就没有意义了，这明显是一个书写错误，后面的 M2 应改为 M4。再往下就没有发现其他错误了。

接着进入动态查错，我们用的第一个例子当然是样例，输入样例后发现输出 7.28，与答案一致。因此没有必要再进行模块测试。现在应该设计自己的测试用例了。

首先使用边界值分析法，这道题在建筑物个数上有两个边界：0 和 20。上边界测使用例设计比较繁琐，因此我们先尝试下边界 0 以及下边界附近的 1。坐标也有两个边界：0 和 1000，我们可以结合建筑物个数的边界，得到如下两个测试用例：

coner1.in

```
0
0 0 1000 1000
```

corner2.in

```
1
0 0 1000 1000
0 0 0 1000 1000 1000
```

对于 corner1 程序输出是 1414.21，正确。对于 corner2 程序输出是 2000.00，也是正确的。接下去我们采用等价分类法设计测试用例。

首先根据数据的排列方式进行分类，我们只使用一个矩形，然后把它的三个顶点的六种排列方式都进行尝试，这样共有 6 个测试用例，下面仅列出一个：

corner3.in

```
1
0 0 1 1
0 0 0 1 1 1
```

答案应该全为 2.00，但测到顶点排列方式为 0 1 1 1 0 0 时，程序输出了 1.41，因为其他条件都不变，因此最有可能便是过程 GetNextPoint 出错。使用 Go to cursor 执行到过程 GetNextPoint 开头处，添加变量数组 P 到 Watch 窗口，再使用 Trace into 一步步执行下去。我们看到程序对 P1、P2 没有进行交换，对 P2、P3 进行了交换，但根据我们的算法，显然需要把（0，0）放到第一个位置。由此恍然大悟，应该在比较完 P1、P2 后，再比较 P1、P3，最后才比较 P2、P3。所以应该在第一句 If-Then-Else 语句后添上：

```
If (P3.x<P1.x)Or((P3.x=P1.x)And(P3.y<P1.y)) Then
  Begin T:=P1;P1:=P3;P3:=T;End;
```

再次检验，答案无误，前面的测试用例也全部通过。我们继续下面的测试。

这次我们根据矩形之间的关系进行分类，两个矩形可以包含，相交，相切，相离。这里的相切是指两个矩形有一部分边重合但不包含，因为题目规定了两个不同建筑物可能十分接近，但永远不会重叠，因此排除相切这种情况，但可以使相离的两个矩形十分接近。这次我们使用两个矩形，有下面三个测试用例：

corner9.in{包含}

```
2
1 0 1 3
0 0 0 3 3 3
1 1 1 2 2 2
```

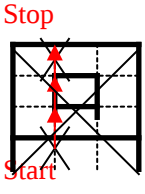
corner10.in{相交}

```
2
1 0 1 1
0 0 0 1 2 1
1 0 1 1 3 1
```

corner11.in{相离，但十分接近}

```
2
1 0 1 1
0 0 0 1 1 1
1.0001 0 1.0001 1 3 1
```

10、11 两个测试用例没有问题，第 9 个输出了 3，有问题，答案应该是 5 才对。我们进入模块调试，数据读入及初始化没有问题，接下去进入 Main 继续跟踪，首先我们对第一个循环用 Go to cursor 一步执行完毕，检查数组 Dis，发现除起点外共有 4 个点的值小于 1e10。因为这是一个矩形包含的用例，按理应该只有两个点的值会小于 1e10。检查这四个点，发现被包含的那个矩形有两个点也被计算在内。仔细一想就明白了，我们虽然加上了两条对角线，但是对于处于矩



形内部的点还是无法排除，我们的程序选择的最短路径应该就是沿直线走，因为中间多了两个矩形顶点把它分为三条线段，这两个顶点又恰好在对角线上，所以这三条线段都不与对角线相交。程序就选择了这条“最短路径”走。
解决方案：因为起点和终点都不会在矩形内部，所以我们可以在读完所有顶点后，排除那些处于矩形内部的点。具体修改可见附录。

我们还可以按照起点和终点的位置关系进行分类，因为这牵涉到矩形的分布，情况比较复杂，所以我们只考虑起点和终点是否重合，因此还需添加一个测试用例：

corner11.in
0
0 0 0 0
输出为 0.00，正确。

最后我们使用错误推测法，这道题中最复杂的无疑是 Intersect 和 InBuilding 这两个函数。因此可以针对这两个函数，把它们分别作为一道小题目来测。可以参照上面讲的步骤设计测试用例，先用边界值分析法，后用等价分类法，具体的测试用例就不再给出。这样测试后的结果也是没有错误。整道题的测试到此就圆满完成了。

【附录】

1、 修改后的程序

```
{说明：红色表示修改后的部分}
Program Corner;
Const
  FinName='corner.in';{输入文件名}
  FoutName='corner.out';{输出文件名}
  MaxBuilding=20;{最大建筑物}
  MaxPoint=82;{最多顶点数}
  MaxLine=120;{最多线段数}
  Zero=1e-8;{相对极小量，用于实数比较}
Type
  Location=Record{坐标类型}
    x,y:Real;
  End;
  Line=Array[1..2]Of Location;{线段类型}
Var
  Bld,{建筑物数}
  Pnt,{顶点数}
  Lne:Integer;{线段数}
  P:Array[1..MaxPoint]Of Location;{顶点序列}
  L:Array[1..MaxLine]Of Line;{线段序列}
  Dis:Array[1..MaxPoint]Of Real;{最短距离表}
Function GetMin(Var a,b:Real):Real;{返回两数较小值}
Begin
  If a>b Then GetMin:=b Else GetMin:=a;
```

```
End;
Function GetMax(Var a,b:Real):Real; {返回两数较小值}
Begin
  If a>b Then GetMax:=a Else GetMax:=b;
End;
Function GetMulti(p0,p1,p2:Location):Real;{计算叉积}
Begin
  GetMulti:=(p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
End;
Function Intersect(Var L1,L2:Line):Boolean;{判断两条线段是否相交}
Var
  M1,M2,M3,M4:Real;
Begin
  Intersect:=False;
  If (GetMin(L1[1].x,L1[2].x)<=GetMax(L2[1].x,L2[2].x))
    And(GetMax(L1[1].x,L1[2].x)>=GetMin(L2[1].x,L2[2].x))
    And(GetMin(L1[1].y,L1[2].y)<=GetMax(L2[1].y,L2[2].y))
       And(GetMax(L1[1].y,L1[2].y)>=GetMin(L2[1].y,L2[2].y)) Then
       {通过快速排斥试验}
    Begin
      M1:=GetMulti(L1[1],L2[1],L1[2]);M2:=GetMulti(L1[1],L1[2],L2[2]);
      M3:=GetMulti(L2[1],L1[1],L2[2]);M4:=GetMulti(L2[1],L2[2],L1[2]);
      If  (Abs(M1)>Zero)And(Abs(M2)>Zero)
      And(Abs(M3)>Zero)And(Abs(M4)>Zero)
      And(M1*M2>0)And(M3*M4>0) Then Intersect:=True;
    End;
End;
Procedure GetNextPoint(Var P1,P2,P3,P4:Location);
{将顶点排序并计算第四个顶点坐标}
Var
  T:Location;
Begin
  If (P2.x<P1.x)Or((P2.x=P1.x)And(P2.y<P1.y)) Then
    Begin T:=P1;P1:=P2;P2:=T;End;
  If (P3.x<P1.x)Or((P3.x=P1.x)And(P3.y<P1.y)) Then
    Begin T:=P1;P1:=P3;P3:=T;End;
  If (P3.x<P2.x)Or((P3.x=P2.x)And(P3.y<P2.y)) Then
    Begin T:=P3;P3:=P2;P2:=T;End;
  P4.x:=P1.x+P3.x-P2.x;
  P4.y:=P1.y+P3.y-P2.y;
End;
Function InBuilding(Var p0:Location):Boolean;{判断点是否在矩形内}
Var
  i,j,k:Byte;
```

```
  M1,M2:Real;
  Temp:Boolean;
Begin
  k:=1;
  For i:=1 to Bld Do
    Begin
      M1:=GetMulti(p0,P[k+4],P[k+1]);
      If Abs(M1)<Zero Then Continue;
      Temp:=True;
      For j:=1 to 3 Do
        Begin
          M2:=GetMulti(p0,P[k+j],P[k+j+1]);
          If (Abs(M2)<Zero)Or(M1*M2<0) Then
           Begin Temp:=False;Break;End;
        End;
      If Temp Then Begin InBuilding:=True;Exit;End;
    End;
  InBuilding:=False;
End;
Procedure Init;{读入数据，并初始化}
Var
  i,j:Byte;
  Mark:Array[1..MaxPoint]Of Boolean;{对要去掉的点坐上标记}
Begin
  ReadLn(Bld);
  Lne:=0;Pnt:=1;
  ReadLn(P[1].x,P[1].y,P[Bld*4+2].x,P[Bld*4+2].y);
  For i:=1 to Bld Do
    Begin
      For j:=1 to 3 Do
        Read(P[Pnt+j].x,P[Pnt+j].y);
      ReadLn;
      GetNextPoint(P[Pnt+1],P[Pnt+2],P[Pnt+3],P[Pnt+4]);
      For j:=1 to 3 Do
        Begin
          L[Lne+j,1]:=P[Pnt+j];
          L[Lne+j,2]:=P[Pnt+j+1];
        End;
      L[Lne+4,1]:=P[Pnt+4];L[Lne+4,2]:=P[Pnt+1];
      L[Lne+5,1]:=P[Pnt+1];L[Lne+5,2]:=P[Pnt+3];
      L[Lne+6,1]:=P[Pnt+2];L[Lne+6,2]:=P[Pnt+4];
      Inc(Pnt,4);Inc(Lne,6);
    End;
  Inc(Pnt);
```

```
  FillChar(Mark,SizeOf(Mark),False);
  For i:=2 to Pnt Do
    If InBuilding(P[i]) Then Mark[i]:=True;
  i:=2;
  While i<Pnt Do
    Begin
      If Mark[i] Then Begin P[i]:=P[Pnt-1];P[Pnt-1]:=P[Pnt];Dec(Pnt);End;
      Inc(i);
    End;
End;
Function GetDis(a,b:Byte):Real;{计算两点间距离}
Begin
  GetDis:=Sqrt((P[a].x-P[b].x)*(P[a].x-P[b].x)+(P[a].y-P[b].y)*(P[a].y-P[b].y));
End;
Function Visible(Var P1,P2:Location):Boolean;
{判断在 P1 点是否能看到 P2 点，即判断两点之间是否有建筑物阻挡}
Var
  TL:Line;
  i:Byte;
Begin
  TL[1]:=P1;TL[2]:=P2;
  For i:=1 to Lne Do
    If Intersect(L[i],TL) Then
      Begin Visible:=False;Exit;End;
  Visible:=True;
End;
Procedure Main;{主程序，用标号法求最短路径}
Var
  i,j,k:Byte;
  Min:Real;
  S:Set Of Byte;
Begin
  Dis[1]:=0;
  For i:=2 to Pnt Do
    If Visible(P[1],P[i]) Then Dis[i]:=GetDis(1,i)
      Else Dis[i]:=1e10;
  S:=[1];
  For i:=2 to Pnt Do
    Begin
      Min:=1e10;k:=0;
      For j:=2 to Pnt Do
        If (Not(j In S))And(Dis[j]<Min) Then Begin k:=j;Min:=Dis[j];End;
      If k=Pnt Then Break;
      S:=S+[k];
```

```
      For j:=2 to Pnt Do
        If (Not(j In S))And(Visible(P[k],P[j]))And(Dis[k]+GetDis(j,k)<Dis[j]) Then
          Dis[j]:=Dis[k]+GetDis(j,k);
    End;
  WriteLn(Dis[Pnt]:0:2);{输出答案}
End;
Begin
  Assign(Input,FinName);
  Reset(Input);
  Assign(Output,FoutName);
  Rewrite(Output);
  Init;
  Main;
  Close(Input);
  Close(Output);
End.
```

二、参考书目

[1]刘福生、王建德。（1993）： 青少年国际信息学（计算机）奥林匹克竞赛指导——人工智能搜索与程序设计。电子工业出版社。

[2]冯树椿、徐六通。（1988）： 程序设计方法学。浙江大学出版社。

[3]向期中 、吴耀斌。（1994）： 信息学（计算机）国际奥林匹克 Turbo Pascal 6.0。中南工业大学出版社。

[4]王建德、柴晓路。（1999）： 国际大学生程序设计竞赛试题解析。复旦大学出版社。