Crafting Winning Solutions

A good way to get a competitive edge is to write down a game plan for what you're going to do in a contest round. This will help you script out your actions, in terms of what to do both when things go right and when things go wrong. This way you can spend your thinking time in the round figuring out programming problems and not trying to figure out what the heck you should do next... it's sort of like precomputing your reactions to most situations.

Mental preparation is also important.

Game Plan For A Contest Round

Read through ALL the problems FIRST; sketch notes with algorithm, complexity, the numbers, data structs, tricky details, ...

- Brainstorm many possible algorithms - then pick the stupidest that works!
- DO THE MATH! (space & time complexity, and plug in actual expected and worst case numbers)
- Try to break the algorithm - use special (degenerate?) test cases
- Order the problems: shortest job first, in terms of your effort (shortest to longest: done it before, easy, unfamiliar, hard)

Coding a problem - For each, one at a time:

- Finalize algorithm
- Create test data for tricky cases
- Write data structures
- Code the input routine and test it (write extra output routines to show data?)
- Code the output routine and test it
- Stepwise refinement: write comments outlining the program logic
- Fill in code and debug *one section at a time*
- Get it working & verify correctness (use trivial test cases)
- Try to break the code - use special cases for code correctness
- Optimize progressively - only as much as needed, and keep all versions (use hard test cases to figure out actual runtime)

Time management strategy and "damage control" scenarios

Have a plan for what to do when various (foreseeable!) things go wrong; imagine problems you might have and figure out how you want to react. The central question is: "When do you spend more time debugging a program, and when do you cut your losses and move on?". Consider these issues:

- How long have you spent debugging it already?
- What type of bug do you seem to have?
- Is your algorithm wrong?
- Do you data structures need to be changed?
- Do you have any clue about what's going wrong?
- A short amount (20 mins) of debugging is better than switching to anything else; but you might be able to solve another from scratch in 45 mins.
- When do you go back to a problem you've abandoned previously?
- When do you spend more time optimizing a program, and when do you switch?
- Consider from here out - forget prior effort, focus on the future: how can you get the most points in the next hour with what you have?

Have a checklist to use before turning in your solutions:

> Code freeze five minutes before end of contest?

- Turn asserts off.
- Turn off debugging output.

Tips & Tricks

- Brute force it when you can
- KISS: Simple is smart!
- Hint: focus on *limits* (specified in problem statement)
- Waste memory when it makes your life easier (if you can get away with it)
- Don't delete your extra debugging output, comment it out
- Optimize progressively, and only as much as needed
- Keep all working versions!
- Code to debug:
    - whitespace is good,
    - use meaningful variable names,
    - don't reuse variables,

        - stepwise refinement,
        - COMMENT BEFORE CODE.
- Avoid pointers if you can
- Avoid dynamic memory like the plague: statically allocate everything.
- Try not to use floating point; if you have to, put tolerances in everywhere (never test equality)
- Comments on comments:
    - Not long prose, just brief notes
    - Explain high-level functionality: ++i; /* increase the value of i by */ is worse than useless
    - Explain code trickery
    - Delimit & document functional sections
    - As if to someone intelligent who knows the problem, but not the code
    - Anything you had to think about
    - Anything you looked at even once saying, "now what does that do again?"
    - Always comment order of array indices
- Keep a log of your performance in each contest: successes, mistakes, and what you could have done better; use this to rewrite and improve your game plan!

Complexity

Basics and order notation

The fundamental basis of complexity analysis revolves around the notion of ``big oh'' notation, for instance: O(N). This means that the algorithm's execution speed or memory usage will double when the problem size doubles. An algorithm of O(N^2) will run about four times slower (or use 4x more space) when the problem size doubles. Constant-time or space algorithms are denoted O(1). This concept applies to time and space both; here we will concentrate discussion on time.

One deduces the O() run time of a program by examining its loops. The most nested (and hence slowest) loop dominates the run time and is the only one mentioned when discussing O() notation. A program with a single loop and a nested loop (presumably loops that execute N times each) is O(N^2), even though there is also a O(N) loop present.

Of course, recursion also counts as a loop and recursive programs can have orders like O(b^N), O($N!$), or even O(N^N).

Rules of thumb

- When analyzing an algorithm to figure out how long it might run for a given dataset, the first rule of thumb is: modern (2004) computers can deal with 100M actions per second. In a five second time limit program, about 500M actions can be handled. Really well optimized programs might be able to double or even quadruple that number. Challenging algorithms might only be able to handle half that much. Current contests usually have a time limit of 1 second for large datasets.
- 16MB maximum memory use
- 2^{10} ~approx~ 10^{3}
- If you have *k* nested loops running about *N* iterations each, the program has O(N^{k}) complexity.
- If your program is recursive with *b* recursive calls per level and has *l* levels, the program O(b^{l}) complexity.
- Bear in mind that there are *N!* permutations and 2^{n} subsets or combinations of *N* elements when dealing with those kinds of algorithms.
- The best times for sorting *N* elements are O(*N* log *N*).
- **DO THE MATH!** Plug in the numbers.

Examples

A single loop with *N* iterations is O(*N*):

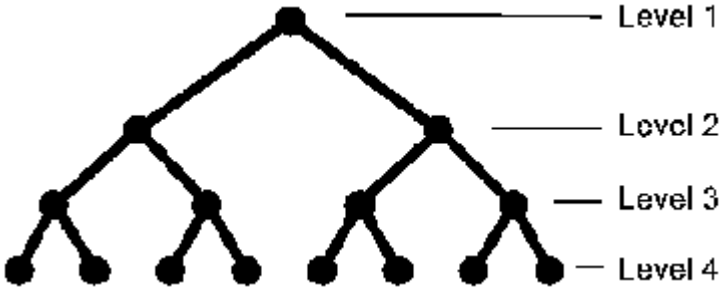
```
1    sum = 0
2    for i = 1 to n
3        sum = sum + i
```

A double nested loop is often O(N^{2}):

```
# fill array a with N elements
1 for i = 1 to n-1
2     for j = i + 1 to n
3         if (a[i] > a[j])
                swap (a[i], a[j])
```

Note that even though this loop executes *N x (N+1) / 2* iterations of the if statement, it is O(N^{2}) since doubling *N* quadruples the execution times.

Consider this well balanced binary tree with four levels:



An algorithm that traverses a general binary tree will have complexity $O(2^N)$.

Solution Paradigms

Generating vs. Filtering

Programs that generate lots of possible answers and then choose the ones that are correct (imagine an 8-queen solver) are *filters*. Those that hone in exactly on the correct answer without any false starts are *generators*. Generally, filters are easier (faster) to code and run slower. Do the math to see if a filter is good enough or if you need to try and create a generator.

Precomputation

Sometimes it is helpful to generate tables or other data structures that enable the fastest possible lookup of a result. This is called *precomputation* (in which one trades space for time). One might either compile precomputed data into a program, calculate it when the program starts, or just remember results as you compute them. A program that must translate letters from upper to lower case when they are in upper case can do a very fast table lookup that requires no conditionals, for example. Contest problems often use prime numbers - many times it is practical to generate a long list of primes for use elsewhere in a program.

Decomposition (The Hardest Thing At Programming Contests)

While there are fewer than 20 basic algorithms used in contest problems, the challenge of combination problems that require a combination of two algorithms for solution is daunting. Try to separate the cues from different parts of the problem so that you can combine one algorithm with a loop or with another algorithm to solve different parts of the problem independently. Note that sometimes

you can use the same algorithm twice on different (independent!) parts of your data to significantly improve your running time.

Symmetries

Many problems have symmetries (e.g., distance between a pair of points is often the same either way you traverse the points). Symmetries can be 2-way, 4-way, 8-way, and more. Try to exploit symmetries to reduce execution time.

For instance, with 4-way symmetry, you solve only one fourth of the problem and then write down the four solutions that share symmetry with the single answer (look out for self-symmetric solutions which should only be output once or twice, of course).

Forward vs. Backward

Surprisingly, many contest problems work far better when solved backwards than when using a frontal attack. Be on the lookout for processing data in reverse order or building an attack that looks at the data in some order or fashion other than the obvious.

Simplification

Some problems can be rephrased into a somewhat different problem such that if you solve the new problem, you either already have or can easily find the solution to the original one; of course, you should solve the easier of the two only. Alternatively, like induction, for some problems one can make a small change to the solution of a slightly smaller problem to find the full answer.

赢得比赛解决方案

翻译 by Google & sacredfantasy

一个好方法获得竞争优势是写下你将要竞赛的计划。这将帮助您策划您的行动，在做什么事情时，都有可能让事情出差错。这样你可以把时间花在你的思想的全面规划问题上，而不是试图找出你应该做的下一个问题... 这有点像你大多数情况下 precomputing 的反应。

心理上的准备也很重要。

一回合的竞赛计划

第一通读所有问题;试着推想到的算法(写下来),复杂性,数字,数据 structs ，微妙的细节， ...

- 集思广益，在许多可能的算法中挑选最简单的!
- 做数学归纳、推理！ （空间和时间复杂度，并推出实际预期和最坏的情况下的数字）
- 试着打破常规的算法-使用特殊方法（退化？ ）测试案例
- 整理问题：最短的工作首先做，在你努力的进程中（最短到最长的：这样做之前，简单，不熟悉，很难）

编写一个程序-对于每一个，一次写一个：

- 最后的算法
- 创建苛刻的测试数据
- 写入数据结构
- 代码输入常规和测试（写出额外输出例程显示的数据？ ）
- 代码的输出常规测试
- 逐步求精：写评论概述程序逻辑
- 填写代码和调试，*一次一个区段*
- 获得工作及正确性验证（使用较小的测试案例）
- 试着打破目前的程序套路-使用特殊的正确的代码
- 逐步优化-只有多达需要，并保持所有版本（使用硬盘的测试情况下，计算出实际运行时）

时间管理的策略和“毁坏控制”的设想

计划各种（可预见的！ ）差错的状况;想象问题，您可能需要和找出您想如何作出反应。 中心问题是： “何时你花更多的时间调试程序，何时你将你的损失排除并继续前进？ ” 。 考虑这些问题：

- 你花费多久调试?
- 你看上去像何种类型的错误?
- 是你的算法错了吗?
- 你的数据结构需要改变?
- 对于你的错误，你有什么线索吗?
- 较短的（ 20 分钟）的调试优于其它任何一种方式;但您可以从零开始编 45 分钟。
- 你什么时候回到问题在您放弃以前?
- 当你花更多的时间优化的程序，你何时开关?
- 从这里考虑出-忘记之前的努力，着眼于未来：你怎么能把握最大的重点在未来的时间里，通过你自己?

有一个清单，然后使用你的解决方案：

比赛的最后五分钟你还在编程?

- 马上结束。
- 关闭调试输出。

提示和技巧

- 您可以暴力搜索
- KISS 原则：简单就是智慧！
- 提示：重点在*时限* （指明的问题）
- 浪费空间当它让您的生活更轻松（如果你能逃脱它）
- 请勿删除您额外的调试输出，注释掉
- 逐步优化，并且尽可能多。
- 保留所有工作版本！
- 代码调试：
    - 空白是好的，
    - 使用有意义的变量名，
    - 不重复使用的变量，
    - 逐步求精，
    - 注释，然后再编号。
- 避免指针如果你可以的话（谁会没事干用这个（ˆ-ˆ））
- 避免动态内存像瘟疫泛滥：静态分配一切。
- 尽量不要使用浮点;如果您要，把公差放在各处（从未测试平等）
- 评论意见：
    - 不长的散文，只是简要说明
    - 解释高层次的功能： ++i; /*增加 I 的值*/不如不用
    - 解释代码权谋
    - 划定及文件功能部分
    - 如某人有智慧知道这个问题，但没有代码
    - 你有什么额外的想法
    - 任何你看着甚至一度表示， “现在是什么做的？ ”
    - 总是评论顺序排列指数
- 保持的记录你的表现在每一个比赛：成功，错误，你可以做得更好;使用此重写和改善你的计划！

复合问题

基础和秩序的符号

根本基础的复杂性分析，围绕着这一概念的``大啊''符号，例如：为 0 （ n ）。这意味着，该算法的执行速度和内存使用量将增加一倍的问题时。 算法为 0 （ n^2 ）将运行速度的 4 倍（或使用 4 倍的空间）的问题时。 恒时间或空间的算法是指为 0 （ 1 ） 。 这一概念适用于时间和空间两个;在这里，我们将集中讨论的时间。

一个推导出的 0 （ ）运行时的程序审查其循环。 最嵌套（因而慢）闭环控制运行时间，并且是唯一一个提到在讨论（ ）符号。程序的单回路和一个嵌套循环（大概循环每执行 *N*次）为 0 （ N^2 ），即使有也是一个为 0 （ n ）环本。

当然，也算作递归循环和递归程序可以有订单像 O(b^N), O(*N!*),或者甚至为 0 （ n^n ）段。

经验法则

- 在分析一个算法，计算出多久，它可能会为给定的数据集，第一个法则是：现代（ 2004 ）计算机可以处理每秒百兆行动。 在五秒时间限制的计划，大约 500M 可处理。 很好的优化程序可以翻一番甚至翻两番，这个数字。具有挑战性的算法可能只能够处理的一半多。 目前竞赛通常有时间限制为 1 秒钟的大型数据集。
- 最高的 16MB 内存使用
- 2^{10} ~约为~ 10^3
- 如果您有 *k* 嵌套循环运行约 *ñ* 反复每个，该计划已经为 O(N^k)的复杂性。
- 如果您的计划是递归递归调用 *与* B 每一级，并 *升*的水平，该计划 O(b^l) 的复杂性。
- 记住，有 *N!*置换和 2^n *不适用*或组合 *N* 在处理这些种算法。
- 最佳时间排序 *N* 元素为 O(*N* log *N*)段。
- **DO THE MATH!**插入的数量。

实例

单循环 *与* N 反复为 0 （ n ） ：

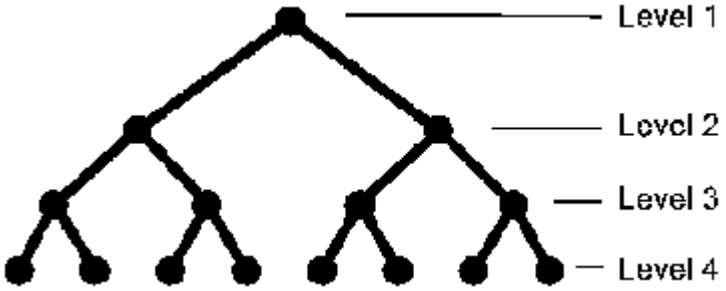
```
1    sum  =  0
2    for  i  =  1  to  n
3        sum  =  sum  +  i
```

双重嵌套循环往往是为 0 （ n^2 ） ：

```
 #  fill  array  a  with  N  elements
1  for  i  =  1  to  n-1
2      for  j  =  i  +  1  to  n
3          if  (a[i]  >  a[j])
                   swap  (a[i],  a[j])
```

请注意，虽然这个循环执行 *N x (N+1) / 2* 迭代的 if 语句，它为 0 （ N^2 ） 自一倍 *ñ* 四倍的执行时间。

审议这个很好的平衡二叉树的四个层次：



一个算法，遍历一般二叉树将复杂性为 O(*2 N*)段。

解决范式

生成 vs 过滤

项目所产生的大量可能的答案，然后选择那些是正确的（想象一个 8 皇后解）的*过滤器* 。那些磨练完全正确的答案没有任何错误的开始是*发电机* 。一般说来，过滤器更容易（快）的代码和运行速度较慢。 做数学，看看是否过滤器是不够好，或者如果您需要尝试，创造一个发电机。

算

有时是有益的生成表格或其他数据结构，使尽可能快地查找的结果。这就是所谓的*算*（在哪一个行业的空间，时间） 。 人们也许可以算数据汇编成一个程序，计算出它的程序启动时，或只记得你的计算结果作为他们。 程序必须把信上以较低的情况下，他们是大写可以做一个非常快速查表无需条件，例如。 比赛的问题经常使用素数-这是多少次的实际构造了一长串素用于其他地方的一个程序。

分解（最难在编程竞赛）

虽然有不少于 20 个基本算法竞赛中使用的问题，面临的挑战相结合的问题，需要结合两种算法的解决方案是艰巨的。 试图分开线索来自不同地方的问题，以使您可以将一个算法循环或与其他算法来解决不同地区的独立问题。 请注意，有时候你可以使用相同的算法在不同的两倍（独立！ ）的部分数据，大大提高您的运行时间。

对称性

很多问题对称性（如之间的距离一分往往是相同的方式，您可以穿越点） 。对称性可 2 路， 4 路， 8 路，等等。 试图利用对称性，以减少执行时间。

例如，与 4 路对称，你只有四分之一解决的问题，然后写下了四个解决方案，份额对称性与单一的答案(寻找自我对称解决方案,这些方案应只输出一次或两次，当然） 。

转寄 vs 落后

令人惊讶的是，许多竞赛工作好得多问题解决倒退时比使用时迎头。 密切注意处理数据顺序相反或建立了攻击，期待中的数据，在一些命令或方式以外的其他显而易见的。

简化

有些问题是可以改写成一个略有不同的问题，例如，如果你解决新问题，您可能已经或可以很容易地找到解决原来的那个，当然，你应该比较容易解决的两个只。另外，像上岗，对一些问题可以作一个小的变化，为解决一个略小问题找到充分的答案。