Optimization

What is Optimization?

Making Working Code Faster

Note *working*. Until a program works, do not try to make it faster, as debugging is then more complicated. Also, before making any change for optimization reasons (instead of debugging reasons), make a backup of the working code. There's nothing more painful then figuring out three optimizations deep that the first optimization broke the program, and no backup exists.

This module will focus on ways to speed programs without altering the program's algorithm. If there's a faster algorithm, that's where effort should be focused, not on making a slow algorithm run a little less slow, which is much like putting perfume on a pig.

In particular, this module will only discuss ways to make a recursive descent program run more quickly, by trying to avoid searching the entire tree. This methodology is referred to as *pruning* the search.

A Problem: Chains

This is a slight adaptation of a real-world problem in compilers, although in that world, only multiplication is available.

Suppose a program must find x^n exactly (no log/exp) and the two standard operations, multiplication and divide, are available. What is the minimal number of these operations required? Show the list (in order) of powers of x that can be calculated in order to find x^n.

For this explication, the calculations will be represented in terms only of the exponent. Multiplication will be written as addition (or the sum of two previous powers), division as subtraction (or the difference of two previous powers). For legal chains, each entry must be either the sum or difference of some two previous entries. So, calculating x^8 would be represented as: 1 2 4 8 (three entries beyond the initial `1'); calculating x^{15} might be represented as: 1 2 4 8 16 15.

Test Set #1

Consider the data set requiring successive calculation of all the possible powers for the exponents 1..50 (not all at once - 50 different test cases). All timings are for a 233 MHz Pentium II.

A Basic Algorithm

Start with the set {1}, perform a depth-first search on generatable numbers (Note that to make this even close to reasonable, the pairs will be chosen in reverse order, that is start with adding the last number in the current sequence to itself, then adding to the previous number, then the difference between the last two in the sequence, etc.)

What's Wrong With This?

It will never terminate.

So, change the algorithm to prune when considering a list that is longer than the best list found thus far. Assume the maximum length chain is 32 steps (a true statement, for exponents up to 65536, but unproven; the assumption will be removed later).

Runtime for sample implementation: 4658.34 seconds.

Optimization Basics

Prune Early, Prune Often

Consider: in a tree of fan-out only 2, getting rid of just two levels everywhere on that tree reduces the number of nodes by a factor of almost 4.

Two basic ideas signal the way to make searching programs faster:

- Don't Do Anything Stupid
- Don't Do Anything Twice

The problem is figuring out what is being done twice and what steps are stupid. All of the optimizations discussed here will utilize some basic fact of the search space. Make sure that the fact is true!

Improving Chain Production

Observation #1

There is one easy way to come up with the numbers, or at least an upper bound. Examine the binary representation of the number. Produce all the powers of 2 up to *n*, and then add up those that that the binary representation suggests. (It is possible to do a little better than this using subtractions, but that won't matter in the long run). Use this scheme to initialize the ``best answer'' data, so time is not spent searching for very long answers.

For example, 43 is 101011 in base 2. Thus, the sequence would start 1, 2, 4, 8, 16, 32. Now, add the ones in the base 2 representation, so 1 + 2 = 3 would be the next number, then 3 + 8 = 11, then 11 + 32 = 43. This yields a sequence of length 9: 1, 2, 4, 8, 16, 32, 3, 11, 43.

Runtime: 4609.81 seconds.

Evaluation: This is just barely OK as optimizations go. It was fairly easy to code, and it did get rid of an unproven assumption, but didn't really buy much in the long run.

Observation #2

Negative numbers are silly. Don't produce them as a new number in the sequence, as that's a ``stupid'' thing to do. Let's say the first negative number in the sequence is the number -42. Obviously -42 was constructed as the difference of two previous elements, so 42 could be constructed just as easily, and it could be used instead of -42 every time 42 was used. Of course, it would be even better if the code were written never to encounter -42.

New runtime: 387.34 seconds

Zero's are even sillier. Creating a silly answer doesn't give you any more ``power.'' Assume the shortest sequence contains a zero. Obviously, if the zero wasn't used in a later operation, it could be dropped from the sequence, resulting a shorter valid solution, so it must have been used. However adding zero and subtracting zero from some value does produces that same value, so any number produced in such a manner could also be dropped. The third alternative, subtract a value from zero results in a negative number, which as noted above, isn't helpful either. Thus, a zero will not be in the shortest sequence.

New runtime: 43.24 seconds

Evaluation: Two simple observations have reduced our runtime by a factor of 100. These are easy to code and give great improvements, the very embodiment of excellent optimizations.

Observation #3

In a single step, the largest possible generate-able integer is exactly double the maximum integer generated so far. Thus, if the maximum achieved so far times 2^(number of steps left until we reach the best found thus far) is less than the goal, there is no hope. Stop now.

Runtime: 0.15 seconds

Evaluation: This was moderately difficult to code, but bought a factor of 300 in runtime, so it was definitely worth it, assuming the contest coding time was available and the problem was running over the time limit.

Test Set Update

The most recent test set runs really quickly, so it's time to make the data set harder. On the new test data set of all powers from 1 to 300, the runtime is 93.21 seconds.

Observation #4

Why not do depth-first search with iterative deepening? The branching factor for this search is very large and grows quadratically as the depth increases, so the additional overhead is very small.

Runtime: 93.05 seconds

Evaluation: In this case, a poor optimization (by itself; things will improve later). It required quite a bit of code change, with a large chance of error, and yielded effectively nothing. This is pretty surprising, as DFSID generally helps immensely.

Observation #5

The most recent operation must use the next-to-last number created. If it didn't, why bother generating that number? (Note that this assumes the DFSID algorithm.)

Runtime: 7.47 seconds

Observation #6

Never duplicate a number in the list.

Runtime: 3.40 seconds

Status Check

Thus far, other than adding depth-first search with iterative deepening, only the ``Don't Do Anything Stupid'' rule has been used, and and the execution time has decreased by an estimated factor of 842,510. That's decent, but probably can be improved.

Test set update

New data set of 1 to 500. Runtime: 206.71 seconds

Observation #7

If a number is to be placed at the end of a sub-prefix chain that is searched completely finding no answer, then adding that same number later doesn't help. For example, if 1 2 4 8 7 ... doesn't work,there's no reason to try 1 2 4 8 16 7 ... later.

Runtime: 53.70 seconds

Observation #8

If the first number selected is i, and the largest number in the sequence is j, then if i + j is less than the minimum next number needed, there's no way to produce it using i.

Runtime: 44.52 seconds

Observation #9

If there a chain that produces x (where x is not the goal) in j steps, but there exists a sequence or length smaller than j which produces that same number, we can just replace that sequence with the shorter one, and obtain a ``better'' sequence, so any chain starting with this longer chain can't be optimal.

WRONG!

First counterexample: 10,127. This hypothesis yields a chain of 17 steps, when one of 16 steps exists.

This is the risk of optimizations: sometimes, they'll be based on incorrect facts. Make sure that you don't fall into this trap.

Conclusion

Eight optimizations yielded a total improvement factor of around 4 million. Additional changes can improve this beyond 44.52 seconds for 1 to 500. One good implementation takes 1.91 seconds, when limited to 640k of memory (0.85 seconds without). See how fast you can make your program.

最优化

译 By Murphy Shang

什么是最优化?

让程序更快

注意：除非你的程序已经正常工作，不要试图优化它，那样会使调试更复杂。而且，在优化以前，将程序备份。没有比优化后程序崩溃而没有备份再痛苦的事了。

本文集中在无法优化算法的情况下优化程序。如果有更快的算法，应该努力实现它，而不是让一个笨拙的算法稍微快一些，那样跟往猪身上抹香水是一样的。

主要，本文只讨论通过剪枝来对搜索算法优化。

题目:数列

假定必须通过乘除运算找出 x^n 的精确值(不能含 log/exp)。这个过程中需要的中间结果分别是多少？按顺序写出项数最少的数列的所有项。

为了简便，我们只写出它们的指数。乘法将写作两数的和,除法将写作两数的差。 对于一个正确的方案，每一个数都应是前面的数的和或差。所以,计算 x^8 将需要 1 2 4 8；计算 x^{15} 需要 1 2 4 8 16 15。

数据规模 1

所有的幂都不大于 50。运行时间在 Pentium II 233 MHz 上得到。

基本算法

从集合{1}开始,进行深度优先搜索(从以前的数中找出两个,将他们的和或差放入队列,依此类推).

这有什么错误?

它不会停止，因为没有加入停止条件。

所以，对于所有长度已经大于 32 的队列，进行剪枝。

运行时间： 4658.34 秒。

基本优化

剪掉出现过的，剪掉常出现的(原文就是这么罗嗦^-^)

注意:在二叉树的一个分支中连续两次剪枝,结点的数量就会变为四分之一。

两个让程序运行更快的思想：

- Don't Do Anything Stupid
- Don't Do Anything Twice

找出已扩展和没有用的节点。所有的优化都会利用简单的事实。确保它是正确的!

改进的方法

注意 1

这里有一个简单的方法来生成数字，或至少可以计算上界。考虑数字的二进制形式。先计算出所有需要的 2 的幂，然后将需要的相加（当然最后答案有可能是相减，但我们没必要求出最优解，用这个方法求最优解是难以实现的）。它就是目前已知的最优解，所以一些很长的数据就会被剪枝。

例如， 43 化为二进制是 101011。那么,数列的起始应该是 1 2 4 8 16 32。然后，将需要的相加，所以下一个数是 1+2=3，然后是 3+8=11，11+32=43，这样生成了一个长度为 9 的数列 1 2 4 8 16 32 3 11 43。

运行时间: 4609.81 秒.

评价：这刚刚是优化的开始。这个方法实现起来很简单，能剪掉一些结点，但不会起太大作用。

注意 2

生成负数是愚蠢的举动。不要生成它们。假如第一个生成的负数是-42。注意-42 是作为两个数的差产生的，所以 42 也会同时生成，而且在以后的生成中，二者起同样的作用。当然，不生成-42 是最好的。

新的运行时间：387.34 秒

生成 0 也是愚蠢的。生成它不会带来任何作用。剪掉任何包含 0 的结点。注意，加或减 0 都会生成相同的数，所以任何含 0 的结点都应该被剪枝。另外，含 0 的结点一定会生成负数，上面已经说过，这是没用的。所以，它应该被剪枝。

新的运行时间：43.24 秒

评价：两个简单的剪枝使运行时间缩短了 100 倍。它们很容易实现而且起到巨大的作用，是极好的剪枝。

注意#3

在一步中，能够生成的最大的数是目前最大数字的 2 倍。那么，如果目前最大的数乘以 2^(目前的步数被最优解的步数减)小于目标，在最优解的步数中它是得不到目标值的。可以剪枝。

运行时间：0.15 秒

评价：实现有一点难度，但是它使运行时间缩短 300 倍，所以这是值得的，对于一般的时限来说，它都能解决。

增加数据规模

目前程序很快，我们应该增加数据规模了。新的数据中幂不大于 300，运行时间是 93.21 秒。

注意 4

为什么不使用可变下界搜索呢？结点的增长随着层数增长，所以结点的质量越来越小。

运行时间：93.05 秒。

评价：既然这样,从数据看这是一个不强的优化(慢慢会好起来)。它使程序代码产生了很大的改变，出错的机会增加了，还没有带来任何好处。这令我们有点惊讶，它一向有很大作用。

注意 5

最近的操作必须使用 next-to-last 数。如果不用，生成它干什么?（这句话是 The most recent operation must use the next-to-last number created. If it didn't, why bother generating that number，我翻译不通）（现在就呈现出可变下界深度优先搜索算法)。

运行时间: 7.47 秒 。

注意 6

不要复制已有的数。

运行时间:：3.40 秒

检查

这样, 除了可变下深度优先搜索以外，只有“Don't Do Anything Stupid”这一原则被体现了，执行时间减少了约 842,510 倍。这是好的，但也许可以再提高。

增加数据规模

新的数据中幂不大于 500，运行时间是 206.71 秒。

注意 7

如果一个数列的前缀没有得到解，在后面加什么数也没有用。例如，如果 1 2 4 8 7 得不到解，就不用尝试 1 2 4 8 16 7 了。

运行时间： 53.70 秒

注意 8

如果选择了 i,数列中最大的数是 j,而且 i+j 小于下一个需要的数,那就不要选择 i。

运行时间: 44.52 秒。

注意 9

如果一个数列生成 x (x 不是目标)用了 j 项，而且存在另外一个数列用了少于 j 项，我们应该用这个短的数列把它替换掉，获得一个“更好的”数列，这是最佳的。

错误!

第一个反例: 10,127.用这种方法得到的解是 17 项,但是存在一个 16 项的方法。

这就是优化的风险：有时候，会得到错误结论。确保你不会掉入陷阱.

总结

8 个优化使运行时间缩小 400 万倍。可以使不大于 500 的数据在 44.52 秒内出解。有的程序可以在 640k 内存限制的情况下 1.91 秒内出解(无限制时 0.85 秒)。看看你的程序有多快。