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.

USACO Gateway | Comment or Question