Complete Search

The Idea

Solving a problem using complete search is based on the ``Keep It Simple, Stupid'' principle. The goal of solving contest problems is to write programs that work in the time allowed, whether or not there is a faster algorithm.

Complete search exploits the brute force, straight-forward, try-them-all method of finding the answer. This method should almost always be the first algorithm/solution you consider. If this works within time and space constraints, then do it: it's easy to code and usually easy to debug. This means you'll have more time to work on all the hard problems, where brute force doesn't work quickly enough.

In the case of a problem with only fewer than a couple million possibilities, iterate through each one of them, and see if the answer works.

How to evaluate 'number of possibilities'

How does one know how many evaluations of some criterion/criteria must be made to determine if brute force is reasonable? One must have an idea of the "rough number of operations" that will be required to solve the task for the highest possible input value(s).

Such an evaluation is what the "big-O" order notation is all about. You'll see algorithms claiming to be O(N) or O(N log N) or O(N*N). This means that (roughly, in our general case) increasing the size of the input will increase the run time proportional to the argument of O().

Thus, for an O(N) solution, doubling N will double the then number of 'operations' performed. However, for O(N*N), when N is, say, 1000 then the runtime is 1,000,000 times slower than the fundamental simple computation in the middle of the loop. This can add up quickly.

Generally O(N*N) (or larger exponents for N) solutions start to get slow when N gets large, but some programming tasks limit N to a small enough value that N*N isn't so large as to preclude brute force.

The most basic way to determine if your program has an O(N), O(N*N), or O(N*N*N) solution is to find the code that is surrounded by the greatest number of loops (i.e., nested most deeply) and tally the loop indexes. If the loops execute unconditionally, simply multiply the three loop sizes together:

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            [do some computing]
        end
    end
end
```

Three nested loops, all size N, O(N*N*N).

The inside computation in a typical contest problem is generally fairly quick, on the order of 10's or 100's of nanoseconds (billionths), occasionally a few microseconds (millionths). But, if N is 1000, an O(N*N*N) solution multiplies this by a billion, turning nanoseconds of inner loop computation into seconds of runtime, generally too slow.

Recursive solutions must be thought through in parallel lines of thinking to the loop solutions above. Recursive solutions, though, incur the additional overhead of allocating stack space for local variables. Each nested recursion allocates a few or even many variables. By the time you're nesting 10,000 (est.) times deep, you're liable to exceed memory limits. Avoid having unused local variables. Calculate the maximum depth of recursion and make sure you won't use up your entire stack space.

Note that sequential parts of a solution are dominated by the single slowest loop (recursion). Three O(N*N) loops and a single O(N*N*N) loop make for an O(N*N*N) execution speed.

Careful, Careful

Sometimes, it's not obvious that you use this methodology.

Problem: Party Lamps [IOI 98]

You are given N lamps and four switches. The first switch toggles all lamps, the second the even lamps, the third the odd lamps, and last switch toggles lamps 1, 4, 7, 10, ...

Given the number of lamps, *N*, the number of button presses made (up to 10,000), and the state of some of the lamps (e.g., lamp 7 is off), output all the possible states the lamps could be in.

Naively, for each button press, you have to try 4 possibilities, for a total of 4^{10000} (about 10^{6020}), which means there's no way you could do complete search (this particular algorithm would exploit recursion).

Noticing that the order of the button presses does not matter gets this number down to about 10000^4 (about 10^{16}), still too big to completely search (but certainly closer by a factor of over 10^{6000}).

However, pressing a button twice is the same as pressing the button no times, so all you really have to check is pressing each button either 0 or 1 times. That's only 2^4 = 16 possibilities, surely a number of iterations solvable within the time limit.

Problem 3: The Clocks [IOI 94]

A group of nine clocks inhabits a 3 x 3 grid; each is set to 12:00, 3:00, 6:00, or 9:00. Your goal is to manipulate them all to read 12:00. Unfortunately, the only way you can manipulate the clocks is by one of nine different types of move, each one of which rotates a certain subset of the clocks 90 degrees clockwise.

Find the shortest sequence of moves which returns all the clocks to 12:00.

The ``obvious'' thing to do is a recursive solution, which checks to see if there is a solution of 1 move, 2 moves, etc. until it finds a solution. This would take 9^k time, where k is the number of moves. Since k might be fairly large, this is not going to run with reasonable time constraints.

Note that the order of the moves does not matter. This reduces the time down to k^9 , which isn't enough of an improvement.

However, since doing each move 4 times is the same as doing it no times, you know that no move will be done more than 3 times. Thus, there are only 4^9 possibilities, which is only 262,144, which, given the rule of thumb for run-time of more than 10,000,000 operations in a second, should work in time. The brute-force solution, given this insight, is perfectly adequate.

Sample Problems

Milking Cows [USACO 1996 Competition Round]

Given a cow milking schedule (Farmer A milks from time 300 to time 1000, Farmer B from 700 to 1200, etc.), calculate

- The longest time interval in which at least one cow was being milked
- The longest time interval in which no cow is being milked

Perfect Cows & Perfect Cow Cousins [USACO 1995 Final Round]

A perfect number is one in which the sum of the proper divisors add up to the number. For example, 28 = 1 + 2 + 4 + 7 + 14. A perfect pair is a pair of numbers such that the sum of the proper divisor of each one adds up to the other. There are, of course, longer perfect sets, such that the sum of the divisors of the first add up to the second, the second's divisors to the third, etc., until the sum of the last's proper divisors add up to the first number.

Each cow in Farmer John's ranch is assigned a serial number. from 1 to 32000. A perfect cow is one which has a perfect number as its serial. A group of cows is a set of perfect cow cousins if their serial numbers form a perfect set. Find all perfect cows and perfect cow cousins.