Brute Force Algorithm

This is the most basic problem solving technique. Utilizing the fact that computer is actually very fast.

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 may 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.

Example 1: *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 the last switch toggles lamps 1,4,7,10,....*

Given the number of lamps, N, the number of button presses made up(upto 10,000), and the state of some of the lamps(eg. 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.

Example 2: *A group of nine clocks inhabits a 3x3 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 the nine different types of move, each one of which rotates a certain subset of the clocks 90 degree 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 49 possibilities, which is only 262,072, 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.

Optimizing your source code :

Your code must be fast. If it cannot be "fast", then it must at least "fast enough". If time limit is 1 minute and your currently program run in 1 minute 10 seconds, you may want to tweak here and there rather than overhaul the whole code again.

1. 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.

2. Pre-Computation/ Pre-Calculation : Sometimes it is helpful to generate tables or other data structures that enable the fastest possible lookup of a result. This is called Pre-Computation (in which one trades space for time). One might either compile Pre-Computed 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.

3. Decomposition : 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.

4. 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).

5. Solving 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.

Source:

http://www.learnalgorithms.in/#