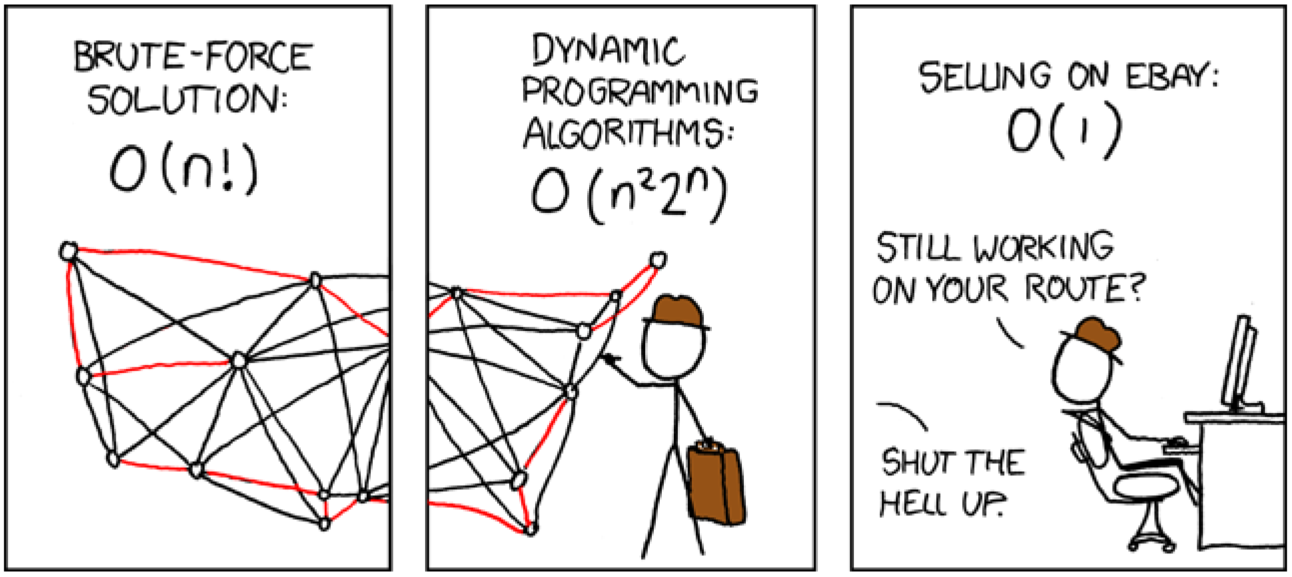
HOW TO ACE AN ALGORITHMS INTERVIEW



Comic courtesy of XKCD, via Creative Commons License

We do a lot of interviewing at Palantir, and let me tell you: it's hard. I don't mean that we ask tough questions (although we do). I mean that the task of evaluating a candidate is hard.

The problem? Given a whiteboard and one hour, determine whether the person across from you is someone you'd like to work with, in the trenches, for the next n years. A candidate's performance during an interview is only weakly correlated with his or her true potential, but we're stuck with the problem of turning the chickenscratch on the whiteboard into an 'aye' or 'nay'. Sometimes it feels like a high-stakes game of reading tea leaves. Believe me we're doing our best, but we're often left with the nagging worry that we're passing up brilliant people who just had a bad day or who didn't click with a particular problem.

In an effort to improve this situation, we wanted to write up a guide that will help candidates make sense of this process, or at least the part known as an Algorithms Interview. At Palantir we ask questions that test for a lot of different skills—coding, design, systems knowledge, etc.—but one of our staple interviews is to ask you to design an algorithm to solve a particular problem.

It usually starts like this:

> Given X, figure out an efficient way to do Y.

First: Make sure you understand the problem. You're not going to lose points asking for clarifications or talking through the obvious upfront. This will also buy you time if your brain isn't kicking in right away. Nobody expects you to solve a problem in the first 30 seconds or even the first few minutes.

Once you understand the problem, try to come up with a solution—any solution whatever. As long as it's valid, it doesn't matter if your solution is trivial or ugly or extremely inefficient. What matters is that you've made progress. This does two things: (1) it forces you to engage with the structure of the problem, priming your brain for improvements you can make later, and (2) it gives you something in the bank, which will in turn give you confidence. If you can achieve a brute force solution to a problem, you've cleared a major hurdle to solving it in a more efficient way.

Now comes the hard part. You've given an O(n^3) solution and your interviewer asks you to do it faster. You stare at the problem, but nothing's coming to you. At this point, there are a few different moves you can make, depending on the problem at hand and your own personality. Almost all of these can help on almost any problem:

1. Start writing on the board. This may sound obvious, but I've had dozens of candidates get stuck while staring at a blank wall. Maybe they're not visual people, but still I think it's more productive to stare at some examples of the problem than to stare at nothing. If you can think of a picture that might be relevant, draw it. If there's a medium-sized example you can work through, go for it. (Medium-sized is better than small, because sometimes the solution to a small example won't generalize.) Or just write down some propositions that you know to be true. Anything is better than nothing.
2. Talk it through. And don't worry about sounding stupid. If it makes you feel better, tell your interviewer, "I'm just going to talk out loud. Don't hold me to any of this." I know many people prefer to quietly contemplate a problem, but if you're stuck, talking is one way out of it. Sometimes you'll say something that clearly communicates to your interviewer that you understand what's going on. Even though you might not put much stock in it, your interviewer may interrupt you to tell you to pursue that line of thinking. Whatever you do, please DON'T fish for hints. If you need a hint, be honest and ask for one.
3. Think algorithms. Sometimes it's useful to mull over the particulars of the problem-at-hand and hope a solution jumps out at you (this would be a bottom-up approach). But you can also think about different algorithms and ask whether each of them applies to the problem in front of you (a top-down approach). Changing your frame of reference in this way can often lead to immediate insight. Here are some algorithmic techniques that can help solve more than half the problems we ask at Palantir:
    - Sorting (plus searching / binary search)
    - Divide-and-conquer
    - Dynamic programming / memoization
    - Greediness
    - Recursion
    - Algorithms associated with a specific data structure (which brings us to our fourth suggestion...)
4. Think data structures. Did you know that the top 10 data structures account for 99% of all data structure use in the real world? Probably not, because I just made those numbers up — but they're in the right ballpark. Yes, on occasion we ask a problem whose optimal solution requires a Bloom filter or suffix tree, but even those problems tend to have a near-optimal solution that uses a much more mundane data structure. The data structures that are going to show up most frequently are:
    - Array
    - Stack / Queue
    - Hashset / Hashmap / Hashtable / Dictionary
    - Tree / binary tree
    - Heap
    - Graph

You should know these data structures inside and out. What are the insertion/deletion/lookup characteristics? (O(log n) for a balanced binary tree, for example.) What are the common caveats? (Hashing is tricky, and usually takes O(k) time when k is the size of the object being hashed.) What algorithms tend to go along with each data structure? (Dijkstra's for a graph.) But when you understand these data structures, sometimes the solution to a problem will pop into your mind as soon as you even think about using the right one.

5. Think about related problems you've seen before and how they were solved. Chances are, the problem you've been presented is a problem that you've seen before, or at least very similar. Think about those solutions and how they can be adapted to specifics of the problem at hand. Don't get tripped up by the form that the problem is presented - distil it down to the core task and see if matches something you've solved in the past.
6. Modify the problem by breaking it up into smaller problems. Try to solve a special case or simplified version of the problem. Looking at the corner cases is a good way to bound the complexity and scope of the problem. A reduction of the problem into a subset of the larger problem can give a base to start from and then work your way up to the full scope at hand. Looking at the problem as a composition of smaller problems may also be helpful. For example, "find a number in a sorted array which has been shifted cyclically by an unknown constant k" can be solved by (1) first figuring out "k" and then (2) figuring out how to perform binary search on a shifted array).
7. Don't be afraid to backtrack. If you feel like a particular approach isn't working, it might be time to try a different approach. Of course you shouldn't give up too easily. But if you've spent a few minutes on an approach that isn't bearing any fruit and doesn't feel promising, back up and try something else. I've seen more candidates who overcommit than undercommit, which means you should (all else equal) be a little more willing to abandon an unpromising approach.

Incidentally, trying out a few different approaches (rather than sticking with a single approach) tends to work well in interviews, because the problems we choose for an interview usually have many different solutions. Happily, the same is true for the problems we solve on the job =)