Above & Beyond CS (ABCS)

Coding Interview Workshop Series

Workshop 2

Design Your Algorithm

Recap: The Goal of a Coding Interview

...is to get signal on things that we do at Facebook every day.

- How you think about and tackle hard problems and how you communicate about code.
  - Evaluate your problem-solving skills to see if you can translate thought into reasonably correct, well-structured code
- How you consider engineering tradeoffs (memory vs. time)
- Limits of what you know

ABCS Tips and Tricks

Before coding

...the first 5'

1. Communicate Proactively ✓
2. Design Your Algorithm
3. Work the Clock

During coding

...the next 10'

4. Writing code at the whiteboard
5. Talk through your code / solution
6. Handling mistakes

"After" coding

...the last 2-3'

7. Test your code
8. Ask questions!

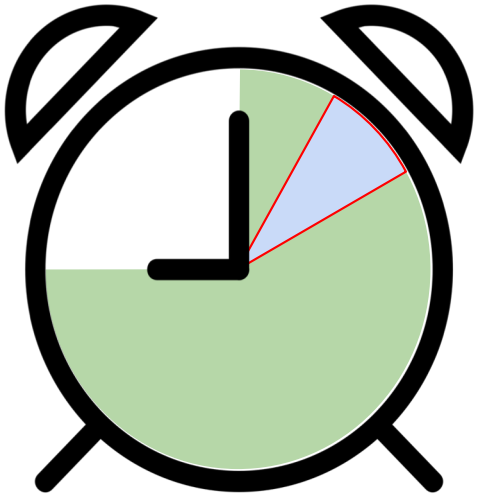
Before you start coding...

1. Communicate proactively
2. Design your algorithm
3. Work the clock

2. Design your algorithm

Design Your Algorithm

There is no sure-fire approach to solving a tricky algorithm problem.



- This is usually the hardest part
- Think about the big picture of the problem first
- Use your time PROACTIVELY and take a minute to think and process your solution approach out loud
  - Make sure that you're not taking 10 minutes to solve it in your head without saying a word

Approach 1: Pattern Matching

Consider what problems the algorithm is similar to, and figure out if you can modify the solution to develop an algorithm for this problem.

Example: A sorted array has been rotated so that the elements might appear in the order 3 4 5 6 7 1 2. How would you find the minimum element?

Similar Problems:

- Find the minimum element in an array.
- Find a particular element in an array (eg, binary search).

Algorithm:

Finding the minimum element in an array isn't a particularly interesting algorithm (you could just iterate through all the elements), nor does it use the information provided (that the array is sorted). It's unlikely to be useful here.

However, binary search is very applicable. You know that the array is sorted, but rotated. So, it must proceed in an increasing order, then reset and increase again. The minimum element is the "reset" point.

Approach 2: Simplify and Generalize

Change a constraint (data type, size, etc.) to simplify the problem. Solve it. Once you have an algorithm for the simplified problem, generalize again.

Example: A ransom note can be formed by cutting words out of a magazine to form a new sentence. How would you figure out if a ransom note (string) can be formed from a given magazine (string)?

Simplification: Instead of solving the problem with words, solve it with characters. That is, imagine we are cutting characters out of a magazine to form a ransom note.

Algorithm:

We can solve the simplified ransom note problem with characters by simply creating an array and counting the characters. Each spot in the array corresponds to one letter. First, we count the number of times each character in the ransom note appears, and then we go through the magazine to see if we have all of those characters.

When we generalize the algorithm, we do a very similar thing. This time, rather than creating an array with character counts, we create a hash table. Each word maps to the number of times the word appears.

Approach 3: Base Case and Build

Solve the algorithm first for a base case (e.g. just one element). Then try to solve it for elements one and two. Then, try to solve it for elements one, two, and three.

Example: Design an algorithm to print all permutations of a string. For simplicity, assume all characters are unique.

Test String: abcdefg

```
Case "a"   --> {a}
Case "ab"  --> {ab, ba}
Case "abc" --> ?
```

This is the first "interesting" case. If we had the answer to P("ab"), how could we generate P("abc"). Well, the additional letter is "c", so we can just stick c in at every possible point. That

```
merge(c, ab) --> cab, acb, abc
merge(c, ba) --> cba, bca, bac
```

Algorithm: Use a recursive algorithm. Generate all permutations of a string by "chopping off" the last character and generating all permutations of s[1… n-1]. Then, insert s[n] into every location of the string.

← this often leads to natural recursive algorithms

Approach 4: Data Structure Brainstorm

This is a bit hacky, but it often works. Simply run through a list of data structures and try to apply each one.

Example: Numbers are randomly generated and stored into an (expanding) array. How would you keep track of the median?

Data Structure Brainstorm:

- Linked list? Probably not – linked lists tend not to do very well with accessing and sorting numbers.
- Array? Maybe, but you already have an array. Could you somehow keep the elements sorted? That's probably expensive. Let's hold off on this and return to it if it's needed.
- Binary tree? This is possible, since binary trees do fairly well with ordering. In fact, if the binary search tree is perfectly balanced, the top might be the median. But, be careful – if there's an even number of elements, the median is actually the average of the middle two elements. The middle two elements can't both be at the top. This is probably a workable algorithm, but let's come back to it.


Don’t let
perfect be
the enemy
of good

Focus on Getting a Working Solution

"Don't let perfect be the enemy of good."

- Acknowledge the brute force solution first
  - Explain the time- and space-complexity of that brute force approach
  - Clarify why this solutions is bad – it's unlikely this is the solution you'll be coding
- Then, consider: "Can I do better?"
  - Think about solutions that optimize for time- or space- complexity; consider tradeoffs
  - Ask the interviewer what you're optimizing for (usually runtime) and narrow down your set of solutions to the one you decide to code
  - Don't worry at first about finding the MOST efficient way to solve the problem
  - Ask yourself: "What's the best way to solve this that I know I can code?"

Consider the Following Scenario

Candidate 1:

- Quickly understands the problem and recognizes the less-efficient solution; knows there's a more elegant approach
- Spends 10 min thinking (in their head) about the best solution approach; doesn't communicate verbally with the interviewer
- Starts writing code but does not arrive at a working solution; does not catch their bugs because there's no time to test code
- Runs out of time to explain the time and space complexity

Candidate 2:

- Does not understand the problem; they've, likely never seen something like this before
- Asks interviewer a few questions, then spends 10 minutes sketching some diagrams and pseudo-code
- Starts writing code for the brute force solution; does not catch their bugs because there's no time to test their code
- Correctly explains the time and space complexity of their solution

Consider the Following Scenario

Put yourself in the interviewer's shoes:

- Do you feel like you have enough signal on how each candidate:
  - Tackles hard problems?
  - Considers engineering tradeoffs?
  - Communicates?
  - The limits of what they know?
- Which candidate appears stronger?
- Would you hire candidate 1, candidate 2, both, or neither?

Communicate Your Design Tradeoffs

After you nail a less-efficient solution that you can get working quickly, then:

- **Even if you only finish your less-efficient solution and then explain how you would do it better, that's not a bad answer!**
- State the time- and space-complexities of your code
  - Address why it's less than ideal (if necessary)
  - You can annotate chunks of your code with their various time and space complexities to demonstrate your understanding of the code
  - Explain any tradeoffs made with regard to time- and space-complexity in your current approach versus alternative approaches

Thank you!

Workshop 3
[INSERT REGION DATE/TIME]

Complete Pre-Work

- ✓ Review screencast
- ✓ Solve HackerRank problems
- ✓ Be prepared to walkthrough your submitted code