Cramming for a tough coding interview?

Dos

1. **Secure as much time as you can**
    1. Move away less important activities from that week
2. **Do targeted preparation for the company you're interviewing for**
    1. Review popular problems from Glassdoor or Careercup or any other sources you can find. Talk to friends who may have interviewed at that company before.
    2. Talk to your recruiter in that company. Ask him/her about the interview process and the things it emphasizes on. That info is not often very precise, but it at least gets some things out of your way e.g. coding language, types of problems you need to focus on etc.
3. **Use resources that are tailored to Coding Interview preparation**
    1. Use a book like Elements of Programming Interviews, or Cracking the Coding interview.
    2. If you're doing it online, use Leetcode. If your company is listed there, start with those problems.
4. **Go Breadth-first**
    1. It's much better to review a few problems in every chapter round robin, than to go deep in any one chapter. That's the advantage of using a book, over a website.
5. **Write code all 7 days**
    1. Write code for one or two problems every day. Use a language that's familiar to you.
6. **Make cheat-sheets and flash cards and use them all 7 days**
    1. Make short cheat-sheets for problems you have done/reviewed. Use them to review problems quickly later. Revision/repetition will strengthen/deepen your understanding of core concepts.
7. **Practice your non-technical parts e.g. your resume projects**
    1. If you have a few years of experience, this will be critical.
    2. Think about what you're going to say about your projects, about yourself , about your career, and about your motivations. If you haven't thought about it at least once, it's very easy to give bad answers under pressure.
8. **Do mock interviews**
    1. One or two is enough, to get the first level of dust off
    2. Do them with experienced engineers who routinely do interviews, instead of with friends or professors or random people off the street.

Don'ts

1. Don't use a theoretical algorithms book

    1. e.g. CLRS is an excellent book, but it's not for one week. You'll get overwhelmed and won't spend time on relevant stuff.
2. Don't skimp on sleep
    1. Intellectually challenging stuff takes time to settle in. Enough sleep is critical, especially the day before the interview.
    2. If you don't sleep well, your brain will work slow, which is the opposite of what you want.
3. Don't try a new programming language
    1. Stick to what you're most comfortable with. Learning a new language in a week, along with prepping for difficult coding interviews is too much to do, and strongly not recommended.
4. Don't practice with a friend, who is not at least as good as you
    1. You can practice with an equally motivated friend if you want to, but you want them to be better than you. If s/he is at least not as familiar as you are, you'll end up wasting each other's time.
5. Don't pin your life's confidence on doing well in this upcoming interview
    1. Interviewing can work with just a little preparation. But it can also be flaky despite best preparation. It's not a standardized test and there is a lot to passing an interview than merely writing code.
    2. If you don't clear this one, get a better strategy for the next one. Don't rest until you've found something you like.
6. Don't jump from book to book or site to site
    1. There is an obscene amount of stuff out there. Pick one or two resources and stick to them.
7. Don't overdo it
8. Don't code for more than 60 to 90 minutes at a stretch.
    1. Be targeted. Whatever you can do, you should do well. That will give you intuition for things you don't have time for.

Here are the main mistakes I see both junior and senior candidates making:

1. **Thinking in their heads instead of on the board.** It's easy to lose track when you're thinking through an edge case or walking through the way variables change as your code executes. **Candidates who draw a picture or write out the variable values see insights and bugs faster and make fewer mistakes.** It might feel like it's too much work to draw it out, but it's worth it. Draw out a sample input to manipulate when you're first figuring out your algorithm, and do it again when you're walking through your code to look for bugs at the end.
2. **Not using a hash table** (or hash, dictionary, array, etc). This is the answer to like half of all interview questions. It should always be your first thought. They have

O(1)-time insertions and lookups (mostly--you should know about hash collisions), so they are a great way to do bookkeeping for many datasets.

3. **Using non-descriptive variable names**. This is likely to confuse you and maybe your interviewer. Be especially careful with loosely-typed languages--consider implying the type in the variable name. It's worth it to write the extra letters on the board each time. And this isn't just to organize your own thinking--it's a strong signal to your interviewer that you write clean, readable code.
4. **Getting stuck in the details before getting the general algorithm down**. This is always hard for me to watch because I happen to be one of those interviewers who doesn't really care about off-by-one errors (I figure you'd sort those out if you were actually at a computer running and testing the code). Leave small optimizations and details for later. Just breeze through the details on the first pass, leave a blank space between each line, and focus on getting your algorithm down. Get it out of your head and onto the board. Write it in English first if you have to (bam! helpful comments). Then walk through it to fill in the details and look for bugs.
5. **Thinking it's a test and not a conversation**. Some candidates talk over their interviewers because they think that getting a hint means they lose points. Many interviewers expect that they'll have to guide you a bit along the way, but *no* interviewers expect that they'll have to fight for your attention. **Your interviewer wants to know what it feels like to work with you. Relax and work together.** Of course, *some* interviewers do think of the interview as more of a test and less of collaboration, but that doesn't mean they won't be irritated when you fail to listen to them or when you try to make yourself look good by saying things like "yeah, that's what I meant" or "I already said that." And please, *don't* be that person who acts impatient, suggests that the question is too easy, etc. Often the question goes deeper than you think, but more importantly your interviewer would much rather hire someone who's rigorous and curious than someone who's condescending and impatient, even if they're brilliant. **You do not win points for being lightning-fast or acting bored.** Smile. Be curious. Be excited, especially when you're hitting a wall ("Oh man, that's tough. I like this one! Hmm....")

Chitchat like a pro.

Before diving into code, most interviewers like to chitchat about your background. They're looking for:

- **Metacognition about coding**. Do you think about how to code well?

- **Ownership/leadership**. Do you see your work through to completion? Do you fix things that aren't quite right, even if you don't have to?
- **Communication**. Would chatting with you about a technical problem be useful or painful?

You should have at least one:

- example of an interesting technical problem you solved
- example of an interpersonal conflict you overcame
- example of leadership or ownership
- story about what you should have done differently in a past project
- piece of trivia about your favorite language, and something you do and don't like about said language
- question about the company's product/business
- question about the company's engineering strategy (testing, Scrum, etc)

Nerd out about stuff. Show you're proud of what you've done, you're amped about what they're doing, and you have opinions about languages and workflows.

Communicate.

Once you get into the coding questions, communication is key. A candidate who needed some help along the way but communicated clearly can be even better than a candidate who breezed through the question.

Understand what kind of problem it is. There are two types of problems:

1. **Coding**. The interviewer wants to see you write clean, efficient code for a problem.
2. **Chitchat**. The interviewer just wants you to talk about something. These questions are often either (1) high-level system design ("How would you build a Twitter clone?") or (2) trivia ("What is hoisting in Javascript?"). Sometimes the trivia is a lead-in for a "real" question e.g.,

"How quickly can we sort a list of integers? Good, now suppose instead of integers we had . . ."

If you start writing code and the interviewer just wanted a quick chitchat answer before moving on to the "real" question, she'll get frustrated. Just ask, "Should we write code for this?"

Make it feel like you're on a team. The interviewer wants to know what it feels like to work through a problem with you, so make the interview feel collaborative. Use "we" instead of "I," as in, "If we did a breadth-first search we'd get an answer in O(n)$O(n)$ time." If you get to choose between coding on paper and coding on a whiteboard, always choose the whiteboard. That way you'll be situated next to the interviewer, facing the problem (rather than across from her at a table).

Think out loud. Seriously. Say, "Let's try doing it this way—not sure yet if it'll work." If you're stuck, just say what you're thinking. Say what might work. Say what you thought could work and why it doesn't work. This also goes for trivial chitchat questions. When asked to explain Javascript closures, "It's something to do with scope and putting stuff in a function" will probably get you 90% credit.

Say you don't know. If you're touching on a *fact* (e.g., language-specific trivia, a hairy bit of runtime analysis), don't try to appear to know something you don't. Instead, say "*I'm not sure, but I'd guess $thing, because...*" The *because* can involve ruling out other options by showing they have nonsensical implications, or pulling examples from other languages or other problems.

Slow the eff down. Don't confidently blurt out an answer right away. If it's right you'll still have to explain it, and if it's wrong you'll seem reckless. You don't win anything for speed and you're more likely to annoy your interviewer by cutting her off or appearing to jump to conclusions.

Get unstuck.

Sometimes you'll get stuck. Relax. It doesn't mean you've failed. Keep in mind that the interviewer usually cares more about your ability to cleverly poke the problem from a few different angles than your ability to stumble into the correct answer. When hope seems lost, keep poking.

Draw pictures. Don't waste time trying to think in your head—think on the board. Draw a couple different test inputs. Draw how you would get the desired output by hand. Then think about translating your approach into code.

Solve a simpler version of the problem. Not sure how to find the 4th largest item in the set? Think about how to find the 1st largest item and see if you can adapt that approach.

Write a naïve, inefficient solution and optimize it later. Use brute force. Do whatever it takes to get *some kind* of answer.

Think out loud more. Say what you know. Say what you thought might work and why it won't work. You might realize it actually does work, or a modified version does. Or you might get a hint.

Wait for a hint. Don't stare at your interviewer expectantly, but do take a *brief* second to "think"—your interviewer might have already decided to give you a hint and is just waiting to avoid interrupting.

Think about the bounds on space and runtime. If you're not sure if you can optimize your solution, think about it out loud. For example:

- "I have to at least look at all of the items, so I can't do better than O(n)*O*(*n*)."
- "The brute force approach is to test all possibilities, which is O(n^2)*O*(*n*2)."
- "The answer will contain n^2*n*2 items, so I must at least spend that amount of time."

Get your thoughts down.

It's easy to trip over yourself. Focus on getting your thoughts down first and worry about the details at the end.

Call a helper function and keep moving. If you can't immediately think of how to implement some part of your algorithm, big or small, just skip over it. Write a call to a reasonably-named helper function, say "this will do X" and keep going. If the helper function is trivial, you might even get away with never implementing it.

Don't worry about syntax. Just breeze through it. Revert to English if you have to. Just say you'll get back to it.

Leave yourself plenty of room. You may need to add code or notes in between lines later. Start at the top of the board and leave a blank line between each line.

Save off-by-one checking for the end. Don't worry about whether your for loop should have "<<" or "<=<=." Write a checkmark to remind yourself to check it at the end. Just get the general algorithm down.

Use descriptive variable names. This will take time, but it will prevent you from losing track of what your code is doing. Use names_to_phone_nums_map instead of nums. Imply the type in the name. Functions returning booleans should start with "is_*" Vars that hold a list should end with "s." Choose standards that make sense to you and stick with them.

Clean up when you're done.

Walk through your solution by hand, out loud, with an example input. Actually *write down*what values the variables hold as the program is running—you don't win any brownie points for doing it in your head. This'll help you find bugs and clear up confusion your interviewer might have about what you're doing.

Look for off-by-one errors. Should your for loop use a "<=<=" instead of a "<<"?

Test edge cases. These might include empty sets, single-item sets, or negative numbers. Bonus: mention unit tests!

Don't be boring. Some interviewers won't care about these cleanup steps. If you're unsure, say something like, "Then I'd usually check the code against some edge cases—should we do that next?"

Practice.

In the end, there's no substitute for running practice questions.

Actually write code with pen and paper. Be honest with yourself. It'll probably feel awkward at first. Good. You want to get over that awkwardness now so you're not fumbling when it's time for the real interview.