

How to Screen Software Engineers

www.hackerrank.com



Introduction

The Complete Guide to Crafting Impactful Interview Coding Challenges

When faced with a flood of candidates for a senior individual contributor position, it's really difficult to gauge their skills. Interviewing for different levels is also difficult because seniority is hard to define. It's a multitude of elements, including deep technical skills, leadership qualities and maturity. It's hard to embody the latter soft skills on a resume. Technical skills, on the other hand, can be assessed using programming interview questions.

Traditionally, companies don't ask coding interview questions until after they've screened their resumes or LinkedIn profiles. And they usually ask them to code **manually on whiteboards**. By then, you've already sunk in countless hours of manpower from recruiters and engineering managers. But there are many famous studies proving that resumes are a poor indicator of success (more on this later). It's just how people have always screened people. When we want to hire great engineers, we ask candidates to prove their skills first and foremost through online programming interview questions. For the purpose of this piece, let's refer to these questions as "code challenges."

By implementing code challenges as an early step in the evaluation process, you can partially qualify candidates and pinpoint the candidates on whom you should spend more time evaluating emotional intelligence and other characteristics.

Today, over a thousand companies across industries, including VMware, Box and Capital One, have adopted this technique of using code challenges as the first step in their evaluation process for senior engineers. Since we power the screening process for thousands of candidates daily, we have a lot of insight into what works and what doesn't.

After analyzing thousands of code submissions, and interviewing several directors of engineering and consultants, we created this guide to design and deliver the most optimal code challenges for your potential senior engineering individual contributors. In the process, we explain why fundamentals are critical for all programmers—no matter how experienced. If this 4,000 word in-depth guide is too long, you can always jump ahead if you'd like. But we really recommend investing some time to consider the proven ways of conducting great interviews.

Table of Contents

01 Part I: Why Assessing Fundamental Skills is Important

- a. You Get Qualified Long-Term Team Members, Not Quick Wins
- b. You Build Long-Lasting, Not Fragile Products
- c. You Never Reinvent the Wheel
- d. But the Interview Can't Be "One-Size-Fits-All"
- e. More Companies Should Prepare Candidates

02 Part II: How to Create a Successful Screening Process

- a. Before You Do Anything
- b. Design Impactful Challenges
 - The Challenge Checklist
- d. Set Expectations, Warm Your Candidates
- e. Calibrate After the Screening

Get in touch with us

[Request Demo](#)

[Free Trial](#)

Or visit [HackerRank.com](https://www.hackerrank.com)

01 Why Assessing Fundamental Skills is Important

For nearly as long as companies have hired programmers, managers have asked engineering candidates to solve fundamental algorithm and data structure problems. And for nearly just as long, engineers have debated the validity of these challenges in job interviews.

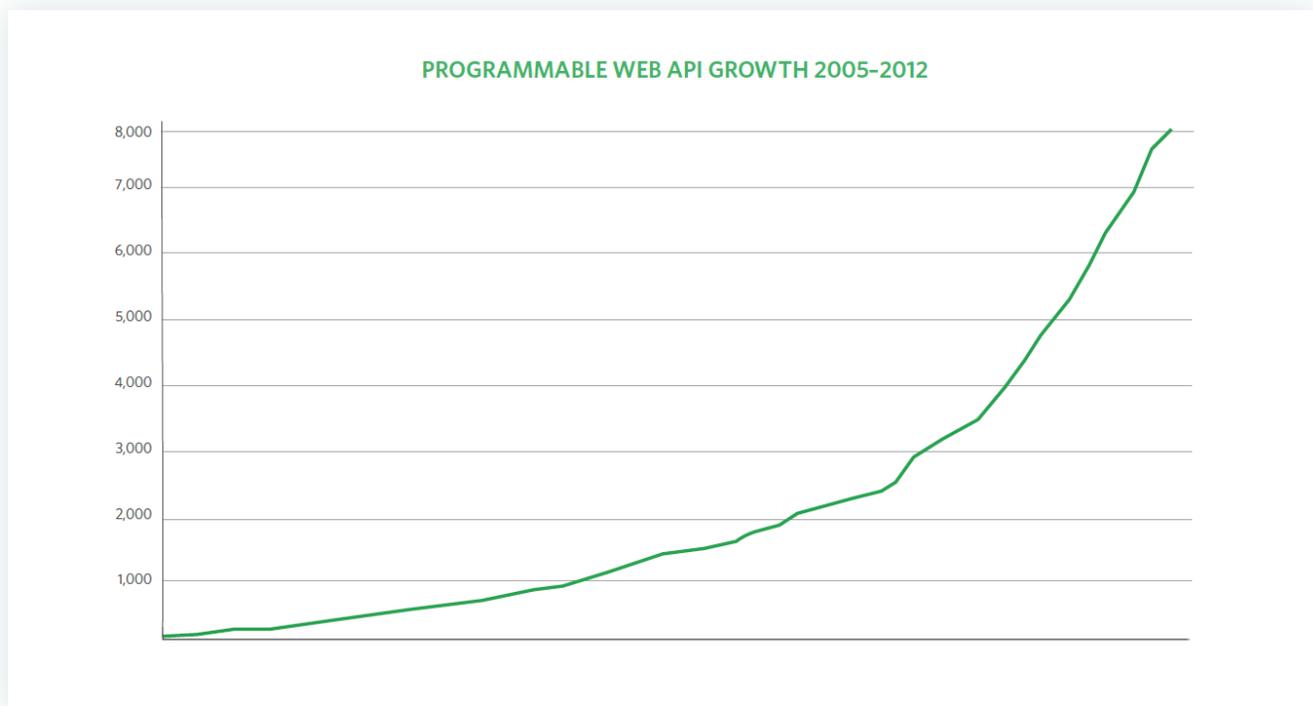
The argument is: If I'm never going to balance a tree on the job, why would you ask these fundamental coding questions to gauge my skillset? At first pass, this can be infuriating for senior engineers. Who's going to remember basic tree-traversal from computer science (CS) courses when you've been using easier, faster standard libraries for years?

But what's not as emphasized as often is the value of basic CS fundamentals for most roles. Everyone knows the best strategy for screening candidates is to test for whatever's important for the job, but simple algorithm questions actually play an important role in uncovering what engineers can and can't do. If you dig deeper, engineers who can't complete basic algorithmic code challenges in an interview are actually less productive hires in the long run.

You Get Qualified Long-Term Team Members, Not Quick Wins

If you don't test for CS fundamentals, you'll risk hiring programmers who are only good at getting things done in the short-term. They can put together decent code using APIs and build a glowing portfolio. But if you ask them why their program works the way it does, they'd see opaque black boxes. It's like they're assembling parts together without a toolkit.

Over the past several years, there's been a sharp boost in the number of APIs and standard libraries. For instance, Salesforce, alone, has over 3 million applications in its third-party app system. Look at the sharp rise in APIs in the last 10 years, according to the ProgrammableWeb.



The uptick of these neat packages make it easy for programmers to get by without revisiting the fundamentals. And that's fine if you just want to hustle, get a quick win and build a stunted product.

But most—if not all—accomplished programmers, from Donald Knuth to Ken Thompson, [value the importance of knowing why code works](#) in building revolutionary products. For instance, Knuth's 1968 masterpiece, *The Art of Programming*, was the first time coders could understand why algorithms work the way they do. "So my book sort of opened people's eyes: 'Oh my gosh, I can understand this and adapt it so I can have elements that are in two lists at once. I can change the data structure.' It became something that could be mainstream instead of just enclosed in these packages."

Testing for algorithms and data structures also tests for lifelong curiosity. Engineers should be "continually interested in keeping themselves up to speed, in revising the fundamentals and taking on intriguing programming problems. Those are the people I want to work with," says Soham Mehta, CEO and co-founder of Interview Kickstart.

You Build Long-Lasting, Not Fragile Products

If you don't test for CS fundamentals, it's going to be really difficult for you to provide for your growing base of customers. When scaling out architecture, you have to understand how components work on a simpler, more fundamental level before applying them across multiple machines. If your engineers open enough logic-related bugs, you could lose valuable customer information or create bottlenecks, resulting in a slow customer experience.

This happened to Ben Sigelman, an ex-Googler who founded a company called LightStep, which builds monitoring and performance tools for developers of large distributed systems. He recently worked with a well-intentioned engineer who decided to use Redis for scalable, consistent and durable storage. But Redis is best as an in-memory data structure server and does not—and cannot—scale well when placed into its “AOF” consistency mode*. In that configuration, Sigelman says it's much slower and less resilient than true distributed databases that append to cluster-level file systems.



“Formal CS training would have triggered a ‘too good to be true’ alarm, well before [the engineer] deployed it, and irrevocably lost user data in the process.”

- Ben Sigelman, Founder, LightStep

You Never Reinvent the Wheel

If you don't test for CS fundamentals, optimizing your codebase is going to take a lot longer than it should. Opponents argue that smart programmers use standard libraries to save time. Why reinvent the wheel when someone else has already solved this problem for you?

But, remember, we're not asking advanced algorithmic interview questions because engineers will be writing algorithms from scratch on the job. We're testing basic knowledge of fundamentals to ensure engineers are not just relying on other people's code, Stackoverflow or Google. Otherwise, when you need to scale and optimize, you'll waste a lot of time trying to figure out optimal solutions. It's not just about memorizing how to implement algorithms. Learning the trade-offs between algorithms is valuable in boosting efficiency. Simply testing candidates on knowledge of where trees fit in relative to sets or maps or linked lists is valuable in and of itself.

Gayle Laakmann McDowell, author of *Cracking the Coding Interview*, offers a great example of what happens when a senior engineer doesn't revise fundamentals:

"A more senior engineer building a parsing engine might not understand how she can leverage graph theory or trees. She could spend hours reinventing the wheel, only to come up with something less optimal in the end."

It's the same for debugging. The most efficient way to debug requires fundamental knowledge of how components behave with one another. Someone who doesn't really know how things work might put in logging everywhere in hopes of catching errors by trial and error. A better way would be to systematically isolate issues by spotting patterns in the errors. An engineer can only do this if he knows the system and its algorithm.

It's especially important if you're not quite sure which specific tools you'll need. If you're building a long-lasting product, it's crucial to test for timeless fundamentals that will be the foundation of future programs. "The breadthfirst search algorithm, for instance, was invented in 1959 as the solution to the shortest path in a maze, but it's still indirectly important to programmers today through some layer of abstraction," says Dr. Herald Memelli, who oversees all of the code challenges at HackerRank.

"Programming tools come and go, but fundamentals are forever."

The assumption that senior engineers don't need to know CS fundamentals on the job couldn't be further from the truth.

But the Interview Can't Be "One-Size-Fits-All"

Of course, you can't rely on general CS questions—alone—to hire for every role. The coding challenges you select have to be appropriate for the role you need filled, and basic fundamentals are one bar that should be cleared.

Leo Polovets has had a lot of experience in designing great screening processes as the second non-founding engineer at LinkedIn and engineer at Google. He offers a solid example:

"For a back-end candidate, you might give them a problem where they need lists, sets, and hash-maps, and you want to make sure they use the right structure at the right time. For a front-end candidate, a good question might be to ask them to do some basic DOM manipulation. These could be 10–20 line programs, but they'll still reveal a lot about what the person can and cannot do," Polovets says.

More Companies Should Prepare Candidates



Gayle Laakmann Mcdowell
Founder, at CareerCup

The reality is, algorithm and data structure interview questions should be really easy—as long as you have some warning, good prep material and context for what interviewers really want to see.

Algorithmic coding challenges aren't designed to evaluate how well you think on your feet. In fact, if you're pop quizzing your candidates on algorithms, you're most likely turning away really great people who happen to test poorly. The best tech companies are preparing their candidates as much as possible to create a stronger, more successful talent pool.

Facebook invests in teaching an interview prep class for all of their candidates. They realize that senior engineers or folks who are self-taught will need to prepare.

It covers exactly what kind of algorithmic coding challenges they plan to ask and explain why. Of the three phases of Facebook's technical interview, one is called "Ninja," which screens the ability to solve tough coding challenges, like sorting algorithms. Any engineer who applies to Facebook has to do really well on these interviews. It's one of the key reasons why Facebook has a [world-class engineering team](#).

The assumption that you don't need to know CS fundamentals on the job couldn't be further from the truth for most jobs. Well-designed basic algorithm and data structures challenges are a good way to gauge depth of technical skills for sustainable products.

02 How to Create a Successful Screening Process

Step 1: Before you do Anything

"The questions don't really matter. What matters first is a clear understanding of what you need," says Soham Mehta, CEO of [Interview Kickstart](#). Optimize your time by spending 80% to figure out what you need and 20% to craft the challenges.



"The questions don't really matter. What matters first is a clear understanding of what you need."

- Soham Mehta, CEO, Interview Kickstart

The two most common values of great engineers are intelligence and technical knowledge. Algorithm challenges are best used to test the former and knowledge or tool-based challenges are great for the latter.

But what most managers don't realize is it's better to keep these values mutually exclusive. Testing for both at the same time puts far too many constraints, limiting your pool of talent. So, how do you know what you need? It largely depends on the size of your company:

a. Finding Smart Senior Engineers

Large companies, like Google and Facebook, are infamous for their algorithm and data structure challenges. When you have extensive teams, it's better to hire for intelligence than knowledge. You likely have enough engineers to teach newcomers your tech stack. And smart people will be able to learn specific technologies pretty quickly anyway.

This is supported by an [85-year-long organizational research](#) study that concludes: “cognitive ability (or intelligence) tests are the best predictor of success across fields.” Algorithm challenges, including ones that ask you to balance trees, are the programming equivalent of “cognitive ability tests” because they test for reasoning, problem solving and critical thinking skills.

Exhibit 1
Estimated validity and use of predictors of managerial success (simple rankings are in parentheses).

Predictor	Research Findings	Expert Opinion	
	Correlation of predictor to performance (validity)* 1	Rankings	
		Estimates of validity** 2	Frequency of use** 3
Cognitive ability tests	0.53 (1)	8.5 (10)	9.1 (10)
Job tryout	0.44 (2)	8.0 (9)	9.9 (11)
Biographical inventory	0.37 (3)	6.2 (7)	4.1 (4)
Reference check	0.26 (4)	4.1 (3)	3.8 (3)
Experience	0.18 (5)	2.2 (1)	3.1 (2)
Interview	0.14 (6)	3.6 (2)	2.2 (1)
Training and experience ratings	0.13 (7)	5.6 (4)	7.5 (8)
Academic achievement	0.11 (8)	6.0 (6)	6.3 (7)
Education	0.10 (9.5)	5.9 (5)	5.9 (5)
Interest	0.10 (9.5)	6.7 (8)	6.2 (6)
Age	-0.01 (11)	9.2 (11)	8.0 (9)

What’s even more interesting is that the screening credentials commonly found on resumes—like education, age (or experience) and academic achievement—ranked among the worst predictors of success.

Years of hosting code challenges on behalf of high-growth companies reveal similar findings.

“Of the thousands of code challenges that companies like Amazon, VMware and Evernote use, algorithm challenges have produced the most successful candidates.”

- Dr. Heraldo Memelli, Sr. Technical Content Engineering Manager, HackerRank

Some might argue that fundamental challenges are better geared toward junior engineers since they’re unrelated to experience. I mean, why would a senior engineer [need to balance trees](#) at this stage of her career, anyway? But McDowell, Mehta and just about every engineering manager we know stresses that testing senior engineers who value fundamentals is crucial for two reasons.

One, if algorithm challenges are the standard measure for intelligence for junior engineers on your team, your bar has to be consistent for senior engineers as well. Otherwise, you'll hire a mix of smart junior engineers and, well, not-so-smart senior engineers.

Second, David Taylor, head of Sonos engineering, loves asking senior engineers data structure and algorithm questions because "if candidates shy away from that, you have a large red flag." It's a way to filter out folks who feel like they're too good to roll up their sleeves and revisit the basics.

Granted, the burden shouldn't be entirely on the candidates. The onus is also on companies to properly prepare senior engineering candidates to ace these fundamental challenges. We'll cover more on this in step 2.

b. Finding Knowledgeable Talent

Smaller companies—ones that can't afford to wait for senior engineers to learn the tech stack they need—are most likely to focus on knowledge over intelligence. By crafting challenges that reveal technical knowledge in specific tools, you'll attract engineers who can start hacking a new application on day one. So, it's better to focus on knowledge specific challenges if you're a stealthy startup.

Step 2: Designing Impactful Challenges

In Step 1, we made the argument that you need to think about which you value more: intelligence or knowledge. Now we'll get to the core of the actual design of these questions. Broadly speaking, we've found that there are three facets of technical skills you should test for:

- How well does he or she know the fundamentals?
- How deep is his or her technical knowledge?
- How deeply can he or she think about problems?

The first two can be tested well before even interacting with the candidate using code challenges. The third facet is best measured in person. This is the longest step of this guide because we offer detailed examples of each category of challenges, as well as a crucial checklist of biggest mistakes that companies make.

Designing Algorithm Code Challenges

There's one thing you need to account for off the bat: Can they code? It sounds obvious. But for years, notable engineers have repeatedly (2007, 2012 and as recent as 2015) pointed out that "the vast majority of so-called programmers who apply for programming job interviews are unable to write the smallest of programs," says Jeff Atwood, renowned programmer and cofounder of StackOverflow. You can weed these folks out by asking them to solve a basic code challenge. Your first challenge should be light and breezy.

Before asking algorithm questions, Mehta, founder of a technical interview preparation site, includes quick multiple-choice challenges that test for some very basic CS fundamentals. e.g.:

01. Given N distinct numbers, how many subsets can you form?

[Answer: 2^N]

02. Given string of length N, how many permutations of that string exist?

[Answer: $N!$, or simply $N \times (N-1) \times (N-2) \dots \times 1$]

Multiple Choice

Multiple choice code challenges are lightweight and are easier on the candidates. Offer several questions, each with plenty of answer choices. That minimizes the chance of getting correct answers by clicking randomly.

Here's a knowledge-based multiple choice example:

Which of the following properly describes a Loader?

- Loaders make it easy to synchronously load data in an activity or fragment.
- Loaders make it easy to asynchronously load data in an activity or fragment.
- Loaders does not make it easy to asynchronously load data in an activity or fragment.
- None of the above

Here's an algorithm-based multiple choice example:

The time required to search an element in a binary search tree having n elements is:

- $O(1)$
- $O(\log_2 n)$
- $O(n)$
- $O(n \log_2 n)$

Code Completion

Code completion is another lightweight style to consider as the first impression with candidates. A code completion challenge is when candidates have to fill-in-the-blank of a given piece of code. [Some of the greatest programming minds](#) of all time spend a great deal of time reading source code. If you provide most of the code, and ask candidates to fill in the lines, it's an interesting way for candidates to solve a challenge while gauging critical thinking and their ability to read other people's code. Plus, it's often more fun. We initiated a contest series exclusively for code completion for our community of developers recently and it's been one of the most engaging contests we've ever held. [Check out the contest for examples](#) of sentence completion code challenges which will help guide you in creating some of your own.

Designing Knowledge-Based Code Challenges

Unlike fundamental algorithm challenges, knowledge-based code challenges are much more straightforward because they're dependent on knowledge of a particular domain or technology. For instance, challenges based on [Client-Server](#), [Sockets](#), [Multi-threading](#) are good signals of seniority in distributed systems.

We asked Dr. Memelli to offer a great example of a knowledge-based code challenge:

UDS Echo Server
by patilarpith

Problem | Submissions | Leaderboard | Discussions

Your task is to write a UNIX Domain Socket(UDS) server which can accept connection from 'N' clients. Each client will send text data over the socket. Read the data and send it back to the client. The server should be Multi-Threaded and should service all clients in parallel. To support Multi-Threading you can use the POSIX thread library (C,C++) or the default threading library for other languages.

Communication Protocol
Read data from client and send the response back.
String "END" marks end of communication from a client. Send response "END" and disconnect the client.

Example request 1:

```
Client 1:
Hello World
END
```



While intelligence-based challenges are the best predictors of success, there's one major differentiator between novice and an expert.

Designing Challenges that Gauge Depth of Thinking

And **it's not the years of experience**. Rather, it's how deeply can you think about problems?

Taylor defines experts as engineers who have proven ownership or thought leadership of large systems, like services, apps or frameworks. With this experience of ownership and tools, experts are scientifically proven to intuitively look at problems in deeper context than novice programmers.

There are quite a few studies that support this:

Skill Level \ Mental Function	NOVICE	COMPETENT	PROFICIENT	EXPERT	MASTER
Recollection	Non-situational	Situational	Situational	Situational	Situational
Recognition	Decomposed	Decomposed	Holistic	Holistic	Holistic
Decision	Analytical	Analytical	Analytical	Intuitive	Intuitive
Awareness	Monitoring	Monitoring	Monitoring	Monitoring	Absorbed

Dreyfus & Dreyfus break down the levels of skill versus mental function.

Novice: “Novices operate from an explicit rules and knowledge-based perspective. They are deliberate and analytical, and therefore slower to take action, they decide or choose.”

Expert: Experts operate from a mature, holistic well-learned understanding, intuitively and without conscious deliberation. This is a function of experience. They do not see problems as one thing and solutions as another, they act.

A second fascinating study similarly uses fMRI technology to measure blood flow in the brain, comparing a novice artist and an expert artist. Again, the findings were in line with what we've seen so far:

Novice: Process information in the area that deals with features (surface-level).

Expert: Process information in the area that deals with deeper meaning.

“The artist ‘thinks’ portraits more than he ‘sees’ them.”

So, how do you really test for depth in thinking? How do we know if engineers can think about software rather than just see its code? Unlike the first two types of challenges (algorithm and knowledge-based), which can be clear-cut automated challenges, questions that measure depth of thought are usually best accomplished in person. This way, you can start off simple and proceed to make the problem statement more complex as you proceed to work it out.

There are two different types of questions that can help you measure seniority by depth of thought:

a. Open-ended questions

After you have a vetted stream of candidates who pass the questions for technical aptitude, asking open-ended questions about big picture strategy is a good way to gauge depth of thought. Taylor finds success with questions like:

“Tell me something you did that had a big impact in a positive way as a result of time to think and strategize. How about the negative impact?”

“I find that mistakes are made when technical people run straight into the how we build something instead of challenging what the right thing to build is and asking the customer-facing questions on what success looks like,” he says.

Other examples of open-ended questions that are critical to test seniority-level and leadership skills:

- How do you build search for Gmail?
- Describe to me some bad code you’ve read or inherited lately
- When do you know your code is ready for production?

b. Infuse multiple parts in your challenges

Josh Tyler, who runs engineering at [Course Hero](#) and recently authored [Building Great Software Engineering Teams](#), says the most effective way to design challenges for senior folks is to create lots of room for depth in the problem statement.

For instance, here’s an example of a good optimization question:

“You are given many words and you have to find frequencies of each word. Here, simple maps, arrays, lists will not work when a huge number of words are given. Instead, you should go for Trie, an efficient data structure here.”



"Most of our interview questions are designed in a way that has many parts. A junior candidate usually only gets through the first part, whereas senior folks get through the second or third parts."

- [Josh Tyler](#), EVP of Engineering & Design, Course Hero

Greg Badros, who founded [Prepared Mind Innovations](#), would break up this problem into the following parts:

- Tell the candidate that you'll start simple and make it more complex as you work through the problem
- Ask for word frequencies
- Make sure they get the simple map solution without coding it
- Tell them how much RAM they've got and how big the dataset is
- Ask them to estimate how big the process will get for the language they'd write this in
- If they get this far, then ask them to propose an alternative data structure that would be more memory efficient because they're out of RAM
- When they're out of design ideas, have them code as far as they got



"Tell the candidate that you'll start simple and make it more complex as you work through the problem."

- [Greg Badros](#), Founder, Prepared Mind Innovations

The Challenge Checklist

So far, in Step 1, we've been focusing on the diverse categories of challenges, based on what you need. Now let's get down to the practical ways of designing challenges. We looked at the question banks of various companies to pinpoint the patterns of what makes a code challenge successful. Based on these patterns, here's a checklist of 5 common mistakes to avoid and ensure each challenge will draw the best candidates:

01. Do you have the right answer?

This sounds like a no-brainer. But you'd be surprised as to how many companies just get their own code challenge wrong. McDowell frequently consults with tech companies of all sizes on their hiring process.



"When I've reviewed companies' question banks, about 10% of answers are wrong," McDowell says.

And it's not a matter of carelessness or minor bugs in typing up the solutions. The challenge designer genuinely believed their algorithm was right until they were proven wrong. Take the time to ensure that 100% of your answers are correct. Otherwise, you'll turn away a lot of strong candidates who were actually answering the question right.

02. Are your algorithm challenges challenging enough?

When it comes to algorithm code challenges, McDowell estimates that if 5% of your candidates instantly know the solution to your question, then it's likely to be too easy. Easy challenges cluster the mediocre and the great, which of course makes identifying great developers very difficult. Your challenge should be hard enough to separate the average from the top.

In an algorithm code challenge, you should have at least one challenging question where only about 20% of your candidates solve it perfectly. A good algorithm question would have 2 (or more) different solutions, one more optimized than the other. This leads to 3 tiers of performances:

- **Tier 1:** The best candidates will pass all test cases because they implemented a correct and optimal solution.
- **Tier 2:** The okay candidates will pass most of the test cases with an algorithm that's correct but suboptimal. Their code gives the right output, but times out on the large test cases.
- **Tier 3:** The candidates you don't want to hire won't develop a correct algorithm at all. If you use a platform like HackerRank, for instance, you can evaluate based on the time to complete a test case, taking into account both asymptotic complexity and the lower constraints. But it doesn't impose a time condition on all problems. It's up to you as the problem designer to decide if the efficiency is important.



03. Are your problem statements detailed and tailored to your company?

The more tailored and specific the problem statements are to your industry, the higher the chances of high-quality candidates completing code challenges as part of their application.

We did an experiment in which we compared two similar companies' challenges—one with a generic problem statement and another with a more tailored challenge. We saw a nearly 10% higher completion rate with the latter. Even anecdotally, we consistently see this direct correlation time and time again. Candidates are more drawn to and interested in challenges themed to your mission versus generic, cold challenges.

[VMware](#), for instance, created a host of very detailed, complex problem statements that delved deep into virtualization. You can see the [Logical Hub Controller](#) problem is richly tailored to a typical virtualized datacenter problem they grapple with daily. By replacing resumes with tailored code challenges for two years, [VMware](#):

- Saved engineers hundreds of hours per quarter by only calling candidates who passed the code challenges
- Increased the success rate for candidates who come onsite (compared to manual resume screenings)

Problem statements can be incredibly daunting, but for senior engineers, this should be a telling challenge. Engineers who can identify the right sub-problems, solve smaller problems and then merge them together are more likely to be successful as leaders in your organization. These larger principles of being able to break things down, spot patterns and attack problems systematically are crucial for any senior engineer. In our experience, companies that use laserfocused problem statements end up with a relatively lower completion rate, but incredibly high interview-to-hire rate. This refers back to the section: [Designing Challenges that Gauge Depth of Thinking](#).



04. Did you rule out luck or bias?

If a candidate gets stuck on some aspect of a problem, were they just unlucky? It's important to distinguish between these factors, and the best way to do this is by getting multiple data points.

That is, ask a question that involves multiple hurdles. This way, if a candidate gets stuck on one aspect, you can help them through it and they still have other logical leaps to overcome (and thus additional data points for your evaluation).

For example, consider this question:

Given an array of people, with their birth and death years, find the year with the highest population.

This problem has a bunch of solutions, each of which presents a new hurdle.

- **The 1st hurdle** is coming up with any correct solution. Most candidates should be able to come up with a brute force algorithm (e.g., for each year that a person is alive, walk through all the people to count the number of people alive in that year), but a few won't. If they don't, give them some help and see if this was just a brief lapse in reasoning or a consistent issue.
- **The 2nd hurdle** might be noticing that you don't need to check the same year repeatedly. You can use a hash table to cache this data.
- **A 3rd hurdle** might be noticing that you only need to check the years between the first birth year and the last death year. So a candidate might first get that information, and then proceed with checking that range of years.
- **A 4th hurdle** might be noticing that, actually, you only need to check the first birth year through the last birth year. You don't need to check years in which people only died; they certainly won't have the highest population.



05. Are you sure there aren't any CS jargon words?

And so on, until we arrive at an optimal solution. Great candidates might even leap past several hurdles at once, and that's a great sign. A problem like this has so many hurdles that you can more effectively see if there's a pattern in the candidate's performance.

For good measure, test out questions on your peers first. Ask at least 2–3 engineers to solve the problem before unleashing onto your candidates. This helps you see what sort of hurdles are in the question and what to expect from candidates.

There's a common misconception that you need to have a computer science (CS) degree in order to be a good programmer. In reality, only 40% of working software engineers in one survey said they had a CS degree.

When we wrote about this in [TechCrunch](#), Wavefront CEO, Sam Pullara told us he agreed.

Senior candidates, particularly, are most likely to be unfamiliar with CS textbook words. For instance, avoid dropping terms like “state machine” or “dependency injection”. Likewise, in order to avoid ruling out senior engineers who haven’t taken computer science fundamental classes (or don’t remember them), avoid questions that involve more obscure algorithm knowledge.



“I’ve worked with a lot of great programmers over the years who don’t have CS degrees. Their biggest disadvantage is that they often lack the jargon of computer science.”

- Sam Pullara, CEO of Wavefront & ex-Yahoo & ex-Twitter

Step 3: Set Expectations, Warm your Candidates

In Step 1, you figured out what you want and designed the challenges to help surface engineers who can help you trailblaze. Now you can focus on the most important part of executing your new screening process: practicing tactful communication to engage engineers.

Senior engineers love a good challenge, but not if it’s time-consuming and futile. The truth is, accomplished engineers will scoff at a cold email prompting them to spend 1-2 hours solving a fundamental code challenge for a company they may never hear from again. It’s understandable—why should they? Chances are they’ve already earned their stripes putting in late hours building software that’s reshaping our world one way or another. When designed and delivered haphazardly, automated code challenges will be perceived as insulting.

It’s critical to infuse an element of empathy in the delivery and design of your challenges. Cold emails with no explanation as to what they’ll be asked and why, are a surefire way to lose senior engineers.

Those who make an attempt to make their candidates feel valued while scaling their screening systems will be the winners of today’s competitive war for talented engineers.

There are a few things you can do to mitigate this:

01

Be very clear about what you intend to ask and what you expect. Don't catch them off guard.

This is an important step in ensuring that you're not rejecting great people just because. For instance, explain that we're not trying to quiz them on random things they don't even need to use on the job.

Instead, fundamental binary tree questions help gauge problem solving and critical thinking skills. If you're sending code challenges with no explanation, reasoning or preparation, your screening tool is an arbitrary knowledge test. **You're bound to anger senior engineers.**

02

Calculate the amount of time you need to take the challenges (usually between 45 and 90 minutes) and give them more than enough time to actually complete it.

Communicate this to your candidates and explain that it shouldn't take you more than an hour, but you have X hours to take it. It helps to simply articulate that you recognize that they're extremely busy. Their time is valuable, so there's no intense time limit to submit.

03

Explain why you're even asking them to do this.

For instance, if the code challenge replaces the resume or initial phone screening, let them know. "We do this code challenge so you don't have to spend more time on the phone with multiple engineers."

04

We did a quick analysis of dozens of code challenges and found they consistently lack CS fundamental knowledge. Most senior engineers are not going to have CS fundamental knowledge right off the bat. First of all, **nearly 60%** of working software engineers don't even have a proper CS background in one study. Second of all, even if they did, it was likely too long ago to remember anything that's applicable in the real world.

Step 4: Calibrate After the Screening

In steps 1-3, you strategized on the best types of challenges to tease the engineers you need. But what good is a code challenge screening if you're testing different candidates with different questions? Consistency can make the data you're collecting meaningful. Memelli finds that all too often, companies change the question after it's already gone out to some candidates, ruining your data set. After the initial screening, it's important to mirror your code challenges with the rest of your interviewing rounds.

It's logical—folks who clear the automated code challenges online are more likely to perform well in-person if the questions are consistent. Automated code challenges are the best tool to predict who will perform well in the in-person interview.

“So, target your questions to mirror the style of questions asked in the actual interview as much as possible.”



Choose questions that you've actually asked in on-site interviews so that you can see how candidates might react. This can help you frame and word the question appropriately as well. Of course, this means you'll have to retire those questions from your in-person interviews.

Tying It All Together

Asking senior engineers to revisit fundamentals in an interview sounds outrageous. And it is if you simply send a cold email without any proper preparation. But automated code challenges are the most objective and successful predictors of hiring we have to filter through candidates at scale.

Gauging not only their intelligence but also how much they value fundamentals through algorithm and data structure questions are strong instruments to find the best engineering talent. If you set expectations and are mindful of senior engineers' complicated hubris, you'll have a stronger pool of senior engineers. And you'll be set up with a replicable system to build and scale your engineering team. A set of well-designed mix of questions that test fundamentals, knowledge and depth of thinking, is a good way to weed out candidates who don't have the basic skills you need to build revolutionary software.

HackerRank

Match Every Developer to the Right Job

HackerRank is a technology hiring platform that is the standard for assessing developer skills for over 1,000 companies around the world. By enabling tech recruiters and hiring managers to evaluate talent objectively at every stage of the recruiting process, HackerRank helps companies hire skilled developers and innovate faster.

www.HackerRank.com

Want to learn more?

Request Demo

Free Trial

Or visit HackerRank.com

USA:
+1-415-900-4023

India:
+91-888-081-1222

UK:
+44-208-004-0258

hello@hackerrank.com
www.hackerrank.com