5 interviews each 45 min.
2 medium/hard questions per interview.
==========
##
Google Notes
Onsite interviews

You'll usually meet with four Googlers—some potential teammates and some cross-functional—for about 30 to 45 minutes each.

All candidates will have the chance to highlight strengths in four different areas:

General cognitive ability: We ask open-ended questions to learn how you approach and solve problems. And there's no one right answer—your ability to explain your thought process and how you use data to inform decisions is what's most important.
Leadership: Be prepared to discuss how you have used your communication and decision-making skills to mobilize others. This might be by stepping up to a leadership role at work or with an organization, or by helping a team succeed even when you weren't officially the leader.
Role-related knowledge: We're interested in how your individual strengths combine with your experience to drive impact. We don't just look for how you can contribute today, but how you can grow into different roles—including ones that haven't even been invented yet.
Googleyness: Share how you work individually and on a team, how you help others, how you navigate ambiguity, and how you push yourself to grow outside of your comfort zone.

For software engineering candidates, we want to understand your coding skills and technical areas of expertise, including tools or programming languages and general knowledge on topics like data structures and algorithms. There's generally some back and forth in these discussions, just like there is on the job, because we like to push each other's thinking and learn about different approaches. So be prepared to talk through your solutions in depth. Push your own boundaries and find the best answer—that's probably how you work anyway.

Technical onsite interviews at Google were historically conducted on whiteboards, but to provide a more authentic coding experience that's less time-consuming, we've started to offer laptops for coding interviews in some sites. These Chromebooks have an interview app that lets you choose a coding language of your preference.

Throughout the interview process, feel free to ask your interviewers for clarification to make sure you fully understand their questions. And feel free to interview us, too. Ask questions—about the work, about the team, about the culture—that will help you decide whether the job will be right for you.
==========

Interviews for all roles

Here's our advice to help you be ready for your interview.

Predict the future: You can anticipate 90% of the interview questions you're going to get. "Why do you want this job?" "What's a tough problem you've solved?" If you can't think of any, Google "most common interview questions." Write down the top 20 questions you think you'll get.
Plan: For every question on your list, write down your answer. That will help them stick in your brain, which is important because you want your answers to be automatic.
Have a backup plan: Actually, for every question, write down THREE answers. Why three? You need to have a different, equally good answer for every question because the first interviewer might not like your story. You want the next interviewer to hear a different story and become your advocate.
Explain: We want to understand how you think, so explain your thought process and decision making throughout the interview. Remember we're not only evaluating your technical ability, but also how you approach problems and try to solve them. Explicitly state and check assumptions with your interviewer to ensure they are reasonable.
Be data-driven: Every question should be answered with a story that demonstrates you can do what you're being asked about. "How do you lead?" should be answered with "I'm a collaborative/decisive/ whatever leader. Let me tell you about the time I … "
Clarify: Many of the questions will be deliberately open-ended to provide insight into what categories and information you value within the technological puzzle. We're looking to see how you engage with the problem and your primary method for solving it. Be sure to talk through your thought process and feel free to ask specific questions if you need clarification.
Improve: Think about ways to improve the solution you present. It's worthwhile to think out loud about your initial thoughts to a question. In many cases, your first answer may need some refining and further explanation. If necessary, start with the brute force solution and improve on it — just let the interviewer know that's what you're doing and why.

Practice: Everyone gets better with practice. Practice your interview answers—out loud—until you can tell each story clearly and concisely.
==========

Coding practice: You can find sample coding questions on sites like CodeLab, Quora, and Stack Overflow. The book “Cracking the Coding Interview” is also a good resource. In some sites, you'll have the option to code on either a Chromebook or a whiteboard, to offer a more natural coding environment. (Ask your recruiter what's available so you can practice.) Be sure to test your code and ensure it’s easily readable without bugs. Don’t stress about small syntactical errors like which substring to use for a given method (e.g. start, end or start, length) — just pick one and let your interviewer know.

Coding: You should know at least one programming language really well, preferably C++, Java, Python, Go, or C. You will be expected to know APIs, Object Orientated Design and Programming, how to test your code, as well as come up with corner cases and edge cases for code. Note that we focus on conceptual understanding rather than memorization.
Algorithms: Approach the problem with both bottom-up and top-down algorithms. You will be expected to know the complexity of an algorithm and how you can improve/change it. Algorithms that are used to solve Google problems include sorting (plus searching and binary search), divide-and-conquer, dynamic programming/memoization, greediness, recursion or algorithms linked to a specific data structure. Know Big-O notations (e.g. run time) and be ready to discuss complex algorithms like Dijkstra and A*. We recommend discussing or outlining the algorithm you have in mind before writing code.
Sorting: Be familiar with common sorting functions and on what kind of input data they’re efficient on or not. Think about efficiency means in terms of runtime and space used. For example, in exceptional cases insertion-sort or radix-sort are much better than the generic QuickSort/MergeSort/HeapSort answers.
Data Structures: You should study up on as many data structures as possible. Data structures most frequently used are arrays, linked lists, stacks, queues, hash-sets, hash-maps, hash-tables, dictionary, trees and binary trees, heaps and graphs. You should know the data structure inside out, and what algorithms tend to go along with each data structure.
Mathematics: Some interviewers ask basic discrete math questions. This is more prevalent at Google than at other companies because counting problems, probability problems and other Discrete Math 101 situations surround us. Spend some time before the interview refreshing your memory on (or teaching yourself) the essentials of elementary probability theory and combinatorics. You should be familiar with n-choose-k problems and their ilk.
Graphs: Consider if a problem can be applied with graph algorithms like distance, search, connectivity, cycle-detection, etc. There are three basic ways to represent a graph in memory (objects and pointers, matrix, and adjacency list) — familiarize yourself with each representation and its pros and cons. You should know the basic graph traversal algorithms, breadth-first search and depth-first search. Know their computational complexity, their tradeoffs and how to implement them in real code.
Recursion: Many coding problems involve thinking recursively and potentially coding a recursive solution. Use recursion to find more elegant solutions to problems that can be solved iteratively.

##
yeshwanth notes:
for c++ bractice bit manupulation
first 2 weeks:
- For new grad role high chance they won't ask design questions, but better to practice it for other companies
- cracking the coding interviews-> good start
- Elements of programming interviews.
- pramp -> two mock interviews every day, helps thinking out load
- keep talking, start brute force(5-10 min) built up on top of that
- testing practice: unit test(edge cases, regression testing, corner case testing)
last week:
leetcode + whiteboard