Build
for
everyone

Google Interview Prep Guide Software Engineering - University Graduate

What's a Software Engineer (SWE)?

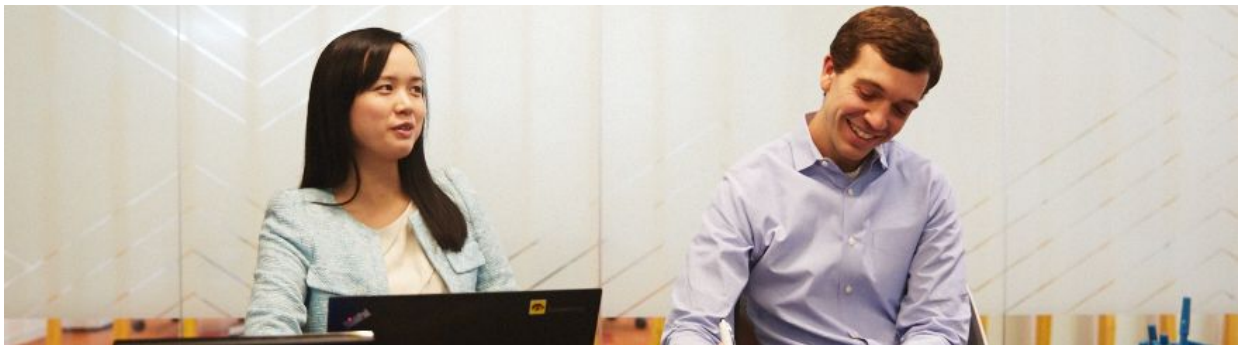
Software Engineers (referred to as "SWEs") at Google develop the next-generation technologies that change how millions of users connect, explore and interact with information and one another. As a SWE, you'll be responsible for the whole lifecycle of a project critical to Google's needs, with opportunities to switch teams and projects as you and our fast-paced business grow and evolve. Depending upon the project you join, you could be involved in research, design, planning, architecture, development, test, implementation and release phases. You'll be working on products that handle information at a massive scale, bringing fresh ideas from all areas and tackling new problems across the full-stack as we continue to push technology forward.

Why Google? Impact.

Google is and always will be an engineering company. We hire people with a broad set of technical skills who are ready to tackle some of technology's greatest challenges and make an impact on millions, if not billions, of users. At Google, engineers not only revolutionize search, they routinely work on massive scalability and storage solutions, large-scale applications and develop entirely new platforms around the world. From AdWords to Chrome, Android to YouTube, Cloud to Maps, Google engineers are changing the world one technological achievement after another.

What to Expect?

Interview topics may cover anything on your resume, whiteboard coding questions, building and developing complex algorithms and analyzing their performance characteristics, logic problems, system design and core computer science principles (hash tables, stacks, arrays, etc.). Computer Science fundamentals are a prerequisite for all engineering roles at Google due to the complexities and global scale of the projects you'll work on.



General Interview Tips

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 solve problems. Explicitly state and check assumptions with your interviewer to ensure they are reasonable.

Clarify - Many 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 - You won't have access to an IDE or compiler during the interview so practice writing code on paper or a whiteboard. 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.

The Technical Phone Interviews

Your phone interview will cover data structures and algorithms. Be prepared to write around 20-30 lines of code in your strongest language. Approach all scripting as a coding exercise — this should be clean, rich, robust code.

1. You will be asked an open-ended question. Ask clarifying questions, devise requirements.
2. You will be asked to explain it in an algorithm.
3. Convert it to a workable code. Hint: Don't worry about getting it perfect because time is limited. Write what comes but then refine it later. Also make sure you consider corner cases and edge cases, production ready.
4. Optimize the code, follow it with test cases and find any bugs.

The Coding & Algorithm Interviews

Sorting - Know how to sort. Don't do bubble-sort. You should know the details of at least one n*log(n) sorting algorithm, preferably two (e.g. quicksort and merge sort). Merge sort can be highly useful in situations where quicksort is impractical, so take a look at it.

Hash Tables - Be prepared to explain how hash tables work and be able to implement one using only arrays in your favorite language in about the space of one interview.

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 Oriented 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.

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.

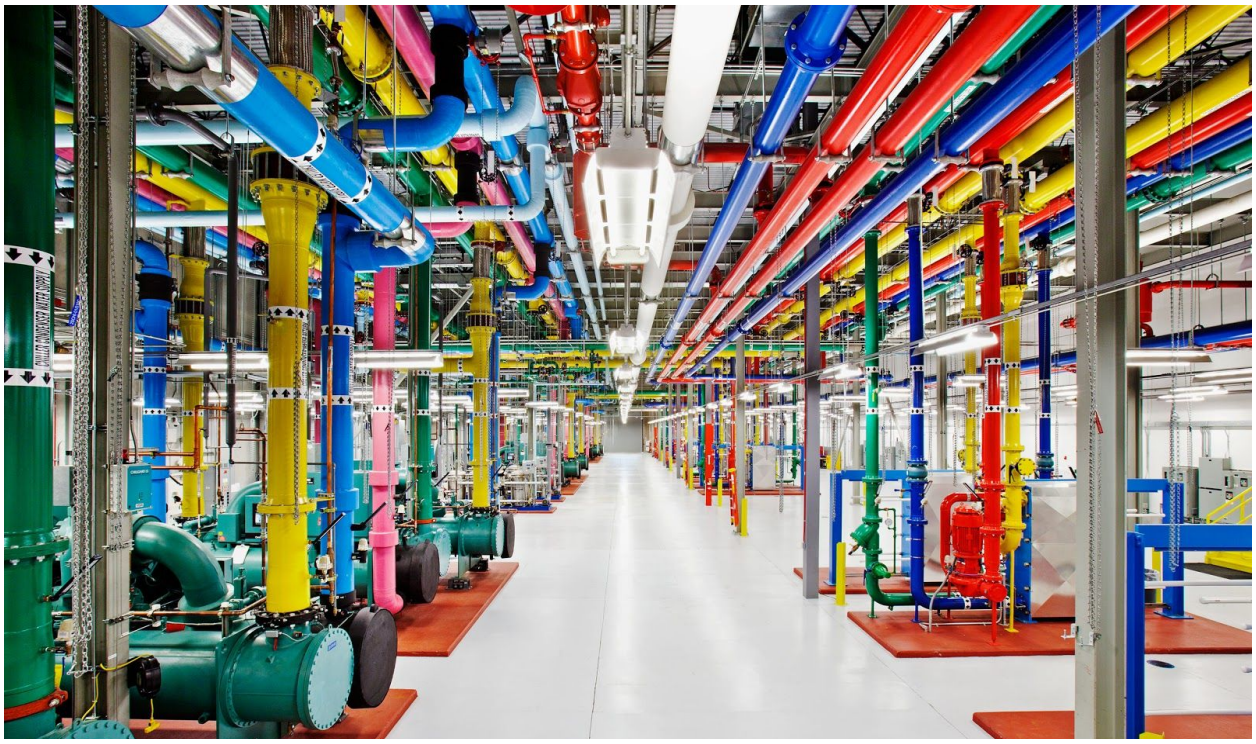
Trees & Graphs - Study up on trees: tree construction, traversal and manipulation algorithms. You should be familiar with binary trees, n-ary trees and trie-trees at the very least. You should know at least one flavor of balanced binary tree, whether it's a red/black tree, a splay tree or an AVL tree and how it's implemented. More generally, there are three basic ways to represent a graph in memory (objects and pointers, matrix and adjacency list), and you should familiarize yourself with each representation and its pros and cons. Understand BFS and DFS tree traversal algorithms and know the difference between inorder, postorder and preorder traversal (for trees). You should be familiar with their computational complexity, tradeoffs and how to implement them in real code. If you get a chance, study up on fancier algorithms such as Dijkstra and A* (for graphs).

Recursion - Many coding problems involve thinking recursively and potentially coding a recursive solution. Prepare for recursion, which can sometimes be tricky if not approached properly. Use recursion to find more elegant solutions to problems that can be solved iteratively.

Operating Systems - You should understand processes, threads, concurrency issues, locks, mutexes, semaphores, monitors and how they all work. Understand deadlock, livelock and how to avoid them. Know what resources a process needs and a thread needs. Understand how context switching works, how it's initiated by the operating system and underlying hardware. Know a little about scheduling and the fundamentals of "modern" concurrency constructs.

System Design - Be able to take a big problem, decompose it into its basic subproblems and talk about the pros/cons of different approaches to solving those subproblems as they relate to the original goal.

Development Practices and Open-Ended Discussion - Sample topics include validating designs, testing whiteboard code, preventing bugs, code maintainability and readability, refactor/review sample code. Other topics: biggest challenges faced, best/worst designs seen, performance analysis and optimization, testing and ideas for improving existing products.



Resources

Books

Cracking the Coding Interview
Gayle Laakmann McDowell

Programming Interviews Exposed: Secrets to Landing Your Next Job
John Mongan, Eric Giguere, Noah Suojanen, Noah Kindler

Programming Pearls
Jon Bentley

Introduction to Algorithms
Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein

Interview Prep

How we hire

Interviewing @ Google

Candidate Coaching Session: Tech Interviewing

CodeJam: Practice & Learn

Technical Development Guide

About Google

Company - Google

The Google story

Life @ Google

Google Developers

Open Source Projects

Github: Google Style Guide

Google Publications

The Google File System

Bigtable

MapReduce

Google Spanner

Google Chubby