



## Google Interview Prep Guide

### Software Engineer

#### 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.



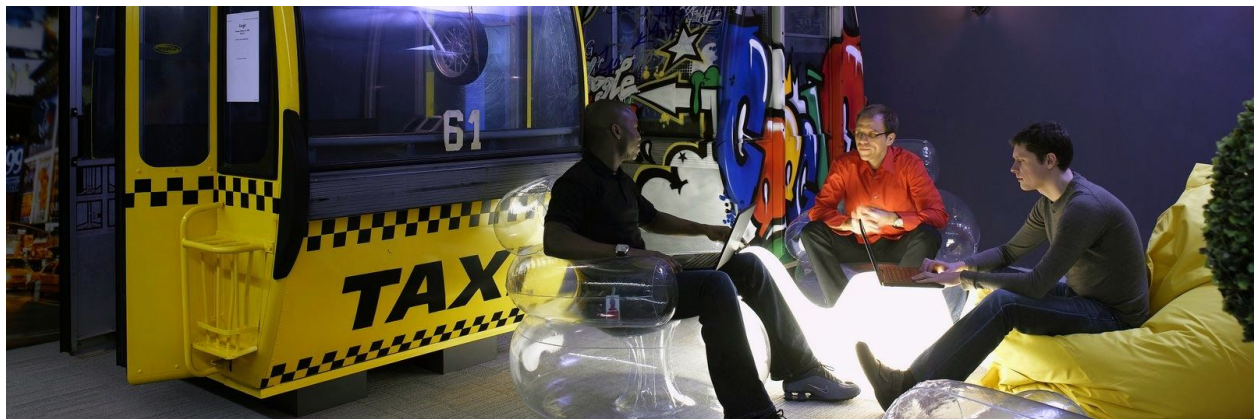
## General Interview Tips

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

**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:** 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

**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.

## The System Design Interviews

**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. We are rapidly moving towards multi-core, so know the fundamentals of "modern" concurrency constructs.

**System Design:** System design questions are used to assess a candidate's ability to combine knowledge, theory, experience and judgement toward solving a real-world engineering problem. Sample topics include features sets, interfaces, class hierarchies, distributed systems, designing a system under certain constraints, simplicity, limitations, robustness and tradeoffs. You should also have an understanding of how the internet actually works and be familiar with the various pieces (routers, domain name servers, load balancers, firewalls, etc.). For information on system design, check out our [research](#) on distributed systems and parallel computing.

## Practice for the Embedded Software Engineering (SWE) Interview

Interviewing for an embedded SWE is very similar to the general SWE process, but there's some nuance in these roles that require a bit of extra preparation. This isn't an exhaustive list of everything that might be covered in your interview, but should give you a good sense of how to prepare.

We recommend reviewing the following chapters of [Cracking the Coding Interview](#) by Gayle Laakmann: Chapter 1 (arrays), Chapter 2 (linked lists), Chapter 3, Chapter 5 (bit manipulation), Chapter 10, Chapter 16 (moderate problems), Chapter 17 (hard problems). Review the concepts and practice solving the coding problems yourself, entering them into a compiler to verify your solutions work and are bug-free. Remember: you won't have the benefit of using a compiler in your interview, so it's important to keep practicing until you can solve problems (bug-free!) in 20 minutes .

**The interview will focus on three main areas:**

- Coding ability
- Understanding of fundamental computer science concepts
- Embedded domain expertise (e.g., compilers, Linux kernel development, bluetooth, low power management, Wi-Fi, storage/server tools, etc.)

**Some fundamental computer science concepts to review** (remember, this isn't an exhaustive list!):

- **Embedded Systems Algorithms/Data Structures:** bit manipulation, arrays, linked lists, mutexes, string manipulation
- **Embedded Systems Design:** constrained resources, computer architecture, User Interface Organizer (UIO)
- **Operating Systems:** Linux, Unix
- **Communications:** input/output (I/O)
- **Storage/Memory:** direct memory access (DMA), access ordering, pointers, caching



## 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](#)