Google Prep – Jimmy

<https://www.quora.com/How-can-I-get-a-job-at-Facebook-or-Google-in-6-months-I-need-a-concise-work-plan-to-build-a-good-enough-skill-set-Should-I-join-some-other-start-up-or-build-my-own-projects-start-up-Should-I-just-focus-on-practicing-data-structures-and-algorithms/answer/Jimmy-Saade?share=1&srid=TAJ1>

[**How can I get a job at Facebook or Google in 6 months? I need a concise work-plan to build a good enough skill set. Should I join some other start-up or build my own projects/start-up? Should I just focus on practicing data structures and algorithms?**](https://www.quora.com/How-can-I-get-a-job-at-Facebook-or-Google-in-6-months-I-need-a-concise-work-plan-to-build-a-good-enough-skill-set-Should-I-join-some-other-start-up-or-build-my-own-projects-start-up-Should-I-just-focus-on-practicing-data-structures-and-algorithms)

This question previously had details. They are now in a comment.

[](https://www.quora.com/profile/Jimmy-Saade)

[Jimmy Saade](https://www.quora.com/profile/Jimmy-Saade), software engineer at Facebook

[Updated Feb 15, 2018](https://www.quora.com/How-can-I-get-a-job-at-Facebook-or-Google-in-6-months-I-need-a-concise-work-plan-to-build-a-good-enough-skill-set-Should-I-join-some-other-start-up-or-build-my-own-projects-start-up-Should-I-just-focus-on-practicing-data-structures-and-algorithms/answer/Jimmy-Saade) · Upvoted by [Uğur Güney](https://www.quora.com/profile/U%C4%9Fur-G%C3%BCney), Data Engineer at Facebook (2017-present)and [Sumeet Ungratwar](https://www.quora.com/profile/Sumeet-Ungratwar), works at Facebook

Some answers here are already really good, but I think it could be valuable to have the perspective from someone who trained for these interviews very recently and got a job offer as a direct result. So I'm gonna one-up your question and tell you how you can get a job at Google and Facebook in 1 month (1 month's prep, that is.) By the way, brevity isn't my strong suit, so this post might take you a while to get through, but I promise it's worth it, and I'll do my best to answer questions you post in the comments about specifics, because I'm almost definitely going to forget to mention some important things (I prepped for the interviews some 5 months ago so this is based on my memory only.)

I'm going to detail how I prepared for technical interviews in ~1 month, after which I got a job at Facebook. The process of getting an interview all the way up to getting an offer will probably take 1-2 months extra after that. For my own experience during the actual interview process, see [Jimmy Saade's answer to What is the software engineering interview process like at Facebook London?](https://www.quora.com/What-is-the-software-engineering-interview-process-like-at-Facebook-London/answer/Jimmy-Saade?share=1&srid=TAJ1) Note that this is for the general Software Engineering position (in my case, new grad), and nothing specific like Android/iOS developer, or Infrastructure Engineer, or so on.

The cool and not-so-convenient thing about tech interviews is that you really never know what you're going to get, so you have to be prepared for a huge range of possible topics, some of which are more likely to occur than others. I'll touch on these below and then outline some very important question-types that may arise and that you should be prepared to deal with.

So let's say your interview is in one month. Here's how I would plan said month (assuming a full-time schedule). Note that this is what I would do (and did, actually), so it might not be the optimum approach for you, but I suggest working similarly and switching it up a bit based on how you feel you'd grasp concepts better (e.g. solve and code in parallel, as opposed to what I did which is solve everything then code everything...)

**Days -∞ to 0 - Prerequisites**

I assume that you have taken an algorithms course and know your way around major data structures including but not limited to: binary trees, binary search trees, hash tables, heaps, stacks, queues, graphs, lists, tries... as well as all algorithms related to them (insert, delete, search, find, find max, find min...) and the time complexity for each of these, at least at a high level. For graphs you need to know searches (BFS and its properties, DFS and its properties including cycle detection and the like) and shortest path algorithms (Dijkstra, Bellman-Ford, and A\*) at a bare minimum. If you don't know all these, along with Dynamic Programming, you're going to need longer than a month. Pick up [Introduction to Algorithms (CLRS)](http://www.amazon.com/Introduction-Algorithms-3rd-Thomas-Cormen/dp/0262033844/ref=sr_1_1?ie=UTF8&keywords=clrs&qid=1432914019&sr=8-1) and start studying them first. (***Update:***I posted an answer here: [**Jimmy Saade's answer to What should I know from CLRS 3rd edition book if my aim is to get into Google?**](https://www.quora.com/What-should-I-know-from-CLRS-3rd-edition-book-if-my-aim-is-to-get-into-Google/answer/Jimmy-Saade?share=1&srid=TAJ1)in regards to which parts of CLRS are relevant for technical interviews.) This is the easy part, as it's all academic and it's just expected that you know all of it. The part that follows below (Day 1 onwards) is the actually valuable part that I can offer you.

I *also* assume that you know a programming language like C++ (or Java) and the built-in functions which actually make it useful (i.e. STL or its Java equivalents). (***Update 2:*** I posted info relevant to this here: [Jimmy Saade's answer to What are the most important concepts in C and C++ that should be learnt and understood before a programming interview?](https://www.quora.com/What-are-the-most-important-concepts-in-C-and-C++-that-should-be-learnt-and-understood-before-a-programming-interview/answer/Jimmy-Saade?share=1&srid=TAJ1)). If you don't know STL, spend time learning vectors, maps, sets, unordered maps, unordered sets, queues, stacks, and the entire "algorithm" library (seriously, all of it). These are essentially implementations of what you just learned in CLRS, so that if you need to use a heap you won't actually start to code one during an interview (just use a map or priority queue). You also need to know how to implement a linked list, BST, and a trie in 5 minutes flat, which is a lot easier than it sounds (just build a Node class and an insert function and for interview purposes, you're good.)

I do **not** assume that you know anything about the following topics: parallel programming, computer networks (HTTP/TCP/IP/Ethernet), operating systems/scheduling, threads/processes/parallelism/concurrency, assembly, hardware and hardware-descriptive languages, or whatever else. While these are all valuable concepts to know as a computer scientist (as are machine learning and AI and others), the chances that they come up are close to none unless you state them as skills on your resume, so your time is better spent elsewhere (i.e. working on the topics below). You do need to have some awareness of distributed computing, though, so scroll down to the System Design section for that and make sure you read the MapReduce paper at the very least.

**Day 1 - The Book**

Buy this book: [Elements of Programming Interviews](http://www.amazon.com/Elements-Programming-Interviews-Adnan-Aziz/dp/1479274836/ref=zg_bsnr_2576_5). Phew. That was hard.  
In all seriousness, this is the best book on the subject in my opinion, and I'm actually really surprised so little people know about it or use it. *(I have no affiliation with this book.)* The collection of questions is excellent and to-the-point, it is large (300+ problems, which is the most I've seen in one book), they focus on the right concepts (e.g. several problems are on binary search, which is *extremely* likely to come up in an interview - more so than any other algorithm), and their answers (and the code provided) are almost all correct and excellent. I say "almost" because there are 1 or 2 problems which have much simpler solutions than the book details, but it's not an issue, especially when you compare it with other programming interview books, which have several answers which are downright incorrect. Plus the online support community is pretty good, with Java code available for all problems (the book has them in C++ only) and an online forum for discussions over at [Home - Elements of Programming Interviews](http://elementsofprogramminginterviews.com/). They also forgo all the 'teaching' stuff that other books have where they try to teach you big-O notation and data structures, and focus almost completely on the problems part, which is much, much, much, much more important. The big-O notation and data structures you should learn from CLRS, which is the best resource for them, period. No other book, *especially* not programming interview books, come close to its quality in teaching that stuff.

I also know (through various sources) that several of these problems are actually asked as-is (or in a disguised form) during interviews, which shows how on-point it is. (I imagine a reason for that may actually be its low popularity compared to other interview books, as companies ban questions that are 'out there' from being asked in interviews, which is why you probably won't see questions from Cracking the Coding Interview.) If this happens to you, however, I suggest you tell your interviewer, as it's very easy for them to tell if you know the problem before or not, and if you just recite the answer it defeats the purpose of the interview. Luckily for me, I wasn't asked any of the problems I'd done from the book.

**Days 2-14 - Algorithms Stage**

Go through the book chapter by chapter, one chapter per day[1], starting at Chapter 5, ending at Chapter 19. Do every single problem. All of them. (To be completely honest, I might've skipped a few, but this was more by accident than anything else, and I definitely did like 98%+ of them.) Don't code, solve the problems only (i.e. find the algorithm). Give yourself a deadline per problem, depending on how hard the problem is (for example, 10 minutes for non-ninja[2] problems, 20 minutes for gray-ninja problems, 30-40 minutes for black-ninja problems) - if you haven't found the solution by then, look at the answer and **understand it.** If you don't you won't improve. It's important to think of the problems on your own, because it's the way of thinking that matters, as you can't go and recite the book on interview day. If you found a solution, make sure it's correct, and that you have thought of all corner cases.

***Note 1:*** The new version of the book (which I linked to) has all the ninja problems in a separate chapter (Ch. 22). This, in my opinion, is a terrible idea. The book I had had the problems which are currently in Ch. 22 spread across the book, each in its relevant chapter. I suggest you go through the relevant ninja problems of each chapter while doing said chapter. For example, on Day 2, do Chapter 5, and the Chapter 5-related problems in Chapter 22. On Day 3, do Chapter 6, and the Chapter 6-related problems in Chapter 22, and so on. I believe the problems in Ch. 22 are ordered accordingly (the ninja problems of Ch. 5 come first, then those of Ch. 6, and so on), so this shouldn't be too hard, but I'm not 100% sure as I have the older copy of the book.  
***Note 2:*** I sometimes spent hours on a single problem, just because I thought the problem was really interesting and I insisted on cracking it myself. I find these random endeavors useful in the long run, as it develops your critical thinking a lot more than the easier problems, but it also takes time, so you likely can't do this for every problem, if you even want to do it at all.

**Days 14-24 - Coding Stage**

Repeat the book, this time with coding. You already know the answers, so you should be able to remember the algorithm for each problem pretty quickly (if you don't, look it up. It happens, and it can happen sometimes even if you'd previously figured the problem out by yourself.) This is the coding stage, so don't waste time re-deriving algorithms.

I do **not** suggest you code all problems, especially if you're experienced with ACM-ICPC, TopCoder, or Codeforces and the like (and really, if you're familiar enough with STL, you probably have a decent skill set). Only write the code for problems you feel have complex algorithms, a new data structure you haven't used before (e.g. unordered map for hashing maybe), problems with tricky corner cases (binary search is at the top of this list as its variants are asked often and can be much trickier than you think) or a programming concept you're not comfortable with (these include, but are not limited to, operator overloading, custom comparators, custom hash functions, custom == functions, and much more...) If a problem proves tricky for you, or you implemented it in a way which you feel isn't optimal, look at the solutions the book provides, which are excellent and clean, and will teach you all of the above-mentioned concepts. I suggest you mimic their style of writing code a bit. Some important-if-obvious notes are: use descriptive variable names (none of that 1-letter-variable-name crap) and indent properly, and don't forget to close parentheses and brackets.

I also suggest you code all problems from the Greedy Algorithms chapter and almost all ninja-marked problems. The Dynamic Programming chapter is also important if you're not familiar with DP, and can be tough to grasp, so make sure you give it its time.

**Day 25 - Onto more questions**

So now that you've exhausted the best question reserve and are comfortable enough to step into an interview, you... need to prep even more. Go to [Google Interview Questions (Career Cup)](http://www.careercup.com/page?pid=google-interview-questions). **This is a dangerous place.**Some very good problems exist, but there's also a class of problems that my ACM trainer likes to call "Chuck Norris problems": Problems written where the OP has no idea what's going on and suggests the interviewer required linear time for problems that clearly cannot be done in linear time (like this, which is clearly not linear time: [http://www.careercup.com/questio...](http://www.careercup.com/question?id=14951746)), or similar.

Now that you've finished Elements of Programming Interviews, you should be easily be able to differentiate between good problems and terrible problems. On Day 25, go through "all" (the last 20 pages or so) the Google Questions (even if you're preparing for Facebook) and make a list of the ones you deem 'good', and by 'good' I mean problems you feel might have actually been asked in a Google interview. You know the question style from the book, so you should be able to tell which are legit and which are questionable. I assume you should have a list of something like 80-120 questions in the end, some simple, some not so much.

Also note that very few problems actually have correct answers posted on the site, so mainly you'll have to rely on your know-how to figure them out and make sure they're correct, but given your previous prep you won't find it too difficult to know when you should be sure of your answer and when you shouldn't. This is actually valuable prep for the actual interview, which is a similar experience.

**Days 26-30 - Solving Career Cup Questions**

Solve all the problems you jotted down on Day 25. Find the algorithm. If you feel it's too difficult, seek help. If you feel it's impossible or the best solution is exponential time, it really might be that the OP was mistaken. Shake it off, move on to another problem. If you still feel like it, code some of the more challenging problems.   
Several of the Career Cup questions are similar to ones in the book, so you shouldn't have too much trouble with most problems.

**Day 30.5 - Skip Lists (Google-only)**

I've heard that Google has recently gotten into the habit of asking about Skip Lists (not sure why). Watch this video:

and understand it and know the analysis of the expected run times. After that, implement and test your very own Skip List. I did this just to practice and because Skip Lists are interesting anyway.

To be honest, Google can be pretty unpredictable with their questions sometimes, in my experience. They might ask general questions about object-oriented programming or computer networking, Linux commands like grep, theoretical things like the proof of the sorting lower bound, coding questions that rely on some math concept you may have forgotten to be solved, or in-depth programming language questions (e.g. functors/operator overloading in C++). I guess it depends on your resume and what you claim to be proficient in, so my advice is not to put anything on there that you’re not at least somewhat proficient in. It helps to have a degree in Computer Science or Electrical and Computer Engineering, really, just based on the huge variety in the possible questions. I suggest a read-through of [Get that job at Google (Steve Yegge)](http://steve-yegge.blogspot.co.uk/2008/03/get-that-job-at-google.html)and [Five Essential Phone Screen Questions (Steve Yegge)](https://sites.google.com/site/steveyegge2/five-essential-phone-screen-questions). You should probably know most of the topics covered here (I wouldn’t put my money on things like threads/processes/parallelism coming up unless you explicitly state it on your resume, though.) Most of the coding questions in the second link are too easy to come up in an interview, I think, so don’t get too excited by them, and I’d skip the “Special Fast Track Version” section. It’s humorous but I thought it’s way too cynical and off-point. Your choice of text editor, knowledge of OS, or knowledge of one vs. multiple languages ***will not***, in and of themselves, make you fail an interview.

On a small note, though I believe Google may ask a lot of non-algorithmic questions as above, the bulk of the interview will still be data structures/algorithms/coding, so all the other things mentioned in Yegge’s blog you should know, but they’re not the main focus.

**Day 31 - The Non-Technical Stuff**

Okay, so I'm cheating a bit by adding Day 31, but you should also take a day or so to prepare for the non-technical part of the interviews, especially if you're interviewing at Facebook, where there's a non-technical interview. First, prepare questions you want to ask your interviewers about Facebook and about their job and what they do all day. See my [Facebook London](https://www.quora.com/What-is-the-software-engineering-interview-process-like-at-Facebook-London/answer/Jimmy-Saade?share=1&srid=TAJ1) post for more examples on this. Second, think over your experiences in college/work/whatever - projects you've worked on, teams you've worked with or managed, conflicts you've addressed, hard bugs you've had to deal with, etc. Google-search "behavioral questions" and you'll find thousands of possible questions.

Prepare a non-generic answer for "Why Facebook" (hint: the fast pace and culture, the great talent in the company, the mission to connect the world...) and "Why Google" (hint: the diversity of the endeavors, the awesomeness of search and Android, the mission to do awesome things, the company culture...). I wasn't asked these questions in either company (to my disappointment since I was really passionate about both and couldn't wait to show it), but I squeezed in my interest while asking my questions to the interviewer, so use that opportunity if you really want to impart something that you didn't get the chance to.

**Tips for the Interviews**

Numbers 3,4,7,8,9 are the most important points.

1. You might be nervous before an interview, but it'll pass. I was nervous before every single interview. Once the interviewer stepped in and we started talking, I generally had a blast because I really loved talking with them and solving these kinds of problems. Try your best not to be too nervous: do mock interviews and the like. I also recommend scheduling interviews in an increasing-priority order, so that you get used to it and find out your shortcomings by the time you reach your most-wanted company.
2. Practice coding without a compiler/on a whiteboard/paper. I did neither, but I have the C++ syntax memorized and I'm used to coding on a paper in ACM competitions, so you might not need to do this if you're already comfortable enough with your favorite language (you only need to know one language well, by the way, as long as it's reasonably well-known, like C++/Java/Python. They let you use whatever language you like during the interview.)
3. Corner cases can kill you. You really have to practice on finding and dealing with corner cases, and/or recognizing what I call "corner-case-prone problems". Some problems are dead simple algorithmically but can be very tricky to code, and I got 2 of these problems, once in my Google phone interviews, and once in my Facebook phone interviews.
4. After finding the algorithm, stop, pause, and think about how to code it, before you actually do. This is especially true for the harder problems, and I would've failed one of my interviews had I not done this, and as a result, would never have gotten a job at FB. I also might've passed an interview at Google which I failed, if I'd taken my advice in this step at the time.
5. Think out loud about algorithms/ideas as you come up with them. It's fine to pause and think quietly for a bit, but don't stand there for 3 minutes without a word. Always at least give the simple solution, which very well might not have a great run-time, but it won't hurt. I did it in all my interviews no matter how simple the answer was, but I said them directly and noted that there's probably a better solution, then proceeded to think of that. (e.g: Okay, to search a sorted array, we can scan it linearly, but this is an O(n) solution and there's likely something faster). Also, don't be cocky about it (question yourself out loud until you're sure of your method and have a rough proof that your method works). Don't argue with your interviewer. 99.99% of the time, they're right, and you're wrong. One possible exception to this is if they’re challenging your code: they’re either really pointing out a bug to you, or trying to make it seem that way to see how confident you are in your code and if you’ll agree blindly or protest that your code is actually correct (if this happens, don’t panic, just think well about your answer before you give it.)
6. Don't talk through your code line by line as you write it. Interviewers know how to read your code and what if-statements and for-loops are. Only speak about the general structure of the code (which you should've mentioned before anyway, as per Tip #4) while coding. Do, however, mention what you're doing in intricate lines of code (for example, if you want to test if 'x' is a power of 2 via "if(x & (x-1))==0", you might want to mention that.)
7. Questions are so often underspecified, and this is a huge weakness of Elements of Programming Interviews: all problems are specified completely, so you have next to no training on this. Always think of questions you might ask or conditions that might make your algorithm fail if not true. Some examples are: Are all numbers positive? Are they distinct? What is the type of the input (integer/double...)? Can you revisit a grid cell? The book has questions where these properties are specified explicitly in the question: think about what would happen if these conditions weren't there: the solution often breaks down.
8. Don't give up if you don't think of the answer directly. In my last Facebook interview, I got the most challenging problem yet, and it took me about 5 minutes to get to the answer, and I ended up hired. That was actually possibly \*the\* interview that got me hired, and it was also the one I most enjoyed.
9. Two really important concepts to know well are binary search (and its variants) and searching the state-space using Breadth-First-Search to find some shortest sequence of 'moves' (like this problem: [ACM-ICPC Live Archive - Kermit the Frog](https://icpcarchive.ecs.baylor.edu/index.php?Itemid=8&category=343&option=com_onlinejudge&page=show_problem&problem=2738)). Both come up very often.
10. Luck matters. The interview process isn't perfect, and you might not pass it even if you're really good, as it depends on your interviewers and what questions you get (and what type of questions you're strong in, etc.) You can mitigate this factor a lot by prepping a huge amount, but it's always there, and it's important to know. I suggest you read [Get that job at Google (Steve Yegge's blog)](http://steve-yegge.blogspot.co.uk/2008/03/get-that-job-at-google.html) if you want some more detail about this factor.
11. Ignore Ch. 20 and 21 in the book. They're not great. (Maybe read through Ch. 21 a bit to get an idea but that's it.) Scroll down to the System Design section if you also have to prepare for a system design interview.
12. Undersell yourself on your CV (or at least, don't oversell yourself), especially if applying through a referral. If you write 'expert in C++', they're going to call up their senior-most C++ engineer to get you to crash and burn. I've never met anyone who got anything related to multithreading and parallelism in an interview for SWE, except one person who listed it as a skill. And lo and behold, he was asked about it, and it didn’t go so well.
13. Oftentimes, you'll get a problem which is a variant of a problem you've seen before in the book or on Career Cup, or is the same problem but in a "disguised form" (i.e. it's worded differently but it has the same or a mostly similar solution.) Be careful about these subtle differences; you might figure out (or *think* that you've figured out) the solution for the problem because you found it very similar to one you've seen before, but a small difference in the problem statement actually means its solution is really really different. As an example, check out question **17.5** - **Search for a sequence in a 2D array** - in Elements of Programming Interviews. It includes the statement *"It is acceptable to visit an entry in A more than once."* With it, the solution is DP. If that statement is not included (i.e. it's not acceptable to visit an entry more than once), the solution is branch-and-bound, and there's no DP involved at all. If you wrongly answer DP instead of branch-and-bound or vice versa, the interviewer will know you've seen the other problem before and think you've just memorized the solution, so that's probably enough by itself to give you a "no-hire" recommendation from that interviewer. (I'd also venture a guess that that statement wouldn't be stated by the interviewer at all first, exactly for this reason, and you'd have to ask whether or not you can visit an entry more than once, as per tip #7. The goal is to see whether or not you'll figure out that there's a huge difference in solutions depending on the interviewer's answer to this question.)

Again, I probably forgot a whole lot of stuff, so if there's anything specific you want to know, leave a comment. I'll also do my best to keep this post updated with whatever other important things I remember later.

**System Design**

***My take on this:***

*Learn the basic concepts and about classical papers, once your thought process is in line with good architects it will be easier to be in sync with the interviewer. Don’t take shortcuts by memorizing a solution. Understand some patterns and common flaws and how to deal with them?*

Even though I didn't have one myself, I did prepare for the System Design interviews. I prepared by visiting this site: [Hired In Tech](http://www.hiredintech.com/system-design), which is decent (not great) and by reading several papers on this site, straight from Google: [Distributed Systems and Parallel Computing](http://research.google.com/pubs/DistributedSystemsandParallelComputing.html), mainly the first MapReduce paper (near the very end of the page) and the Chubby paper. MapReduce is very important and I really suggest you read it and understand how it works. After those steps, look up databases, specifically SQL and [NoSQL](https://www.mongodb.com/nosql-explained), get acquainted with the [CAP theorem](http://en.wikipedia.org/wiki/CAP_theorem), scalability topics, and maybe read up on Hadoop and some problems you can solve with it (Hadoop In Practice is a decent book for these purposes). Try some questions like the "Design a URL shortener" question on Hired In Tech, or something larger scale like "Design a web search engine" or "Design Google Maps", all questions which may be asked (also check Ch. 21 of the book for possible questions and a small idea of how to answer them - though the book's answers aren't great.) But in general, for the system design interview, practicing on questions is less meaningful than fundamentally understanding the above concepts and knowing how to discuss them, as the entire interview is something like a quick conversation between you and the interviewer, where he/she will change the question specifications on the fly to see how you deal with different scenarios.

**Final Advice**

So, if you really want that job, it’s going to take some time and dedication, but hopefully it’s the enjoyable kind. I personally really enjoyed preparing these kinds of questions and found that, job aside, I really learned a lot and got a good deal of knowledge out of the preparation, and you probably will too.

My final piece of advice is to just go into the interview and not be stressed out (this is obviously easier said than done). The engineers *want* you to be good and they *want* to hire you – hiring is a pretty expensive process. Some may be easygoing, and some may be less forgiving, but in all cases, the interview is very similar to a conversation between two engineers, and that’s exactly what these companies strive for the interview to be, so just treat it that way, and if you’ve prepared well, it’ll show.

[1] - One chapter per day is actually a bit slow since you're not coding, so for shorter chapters such as Chapters 5, 7, 8, 9, I suggest you do 2 per day, which is feasible.  
[2] - In Elements of Programming Interviews, non-ninja problems are standard problems, gray-ninja problems are somewhat difficult, and black-ninja problems are difficult.

*Disclaimer: This is my own opinion/advice, and is not endorsed by anyone else in any way.*