your interviewer will make an assessment of your performance, usually based on the following:

- Analytical skills: Did you need much help solving the problem? How optimal was your solution? How long did it take you to arrive at a solution? If you had to design/architect a new solution, did you structure the problem well and think through the tradeoffs of different decisions?
- Coding skills: Were you able to successfully translate your algorithm to reasonable code? Was it clean and well-organized? Did you think about potential errors? Did you use good style?
- Technical knowledge/ Computer Science fundamentals: Do you have a strong foundation in computer science and the relevant technologies?
- Experience: Have you made good technical decisions in the past? Have you built interesting, challenging projects? Have you shown drive, initiative, and other important factors?
- Culture fit/ Communication skills: Do your personality and values fit with the company and team? Did you communicate well with your interviewer?

- SDETs (software design engineers in test) write code, but to test features instead of build features. As such, they have to be great coders and great testers. Double the prep work!

At a very, very high level, there are four modes of questions:

- **Sanity Check**: These are often easy problem-solving or design questions. They assess a minimum degree of competence in problem-solving. They won't tell distinguish between "okay" versus "great'; so don't evaluate them as such. You can use them early in the process (to filter out the worst candidates), or when you only need a minimum degree of competency.
- **Quality Check**: These are the more challenging questions, often in problem-solving or design. They are designed to be rigorous and really make a candidate think. Use these when algorithmic/problem solving skills are of high importance. The biggest mistake people make here is asking questions that are, in fact, bad problem-solving questions.
- **Specialist Questions**: These questions test knowledge of specific topics, such as Java or machine learning. They should be used when for skills a good engineer couldn't quickly learn on the job. These questions need to be appropriate for true specialists. Unfortunately, I've seen situations where a company asks a candidate who just completed a 10-week coding bootcamp detailed questions about Java. What does this show? If she has this knowledge, then she only learned it recently and, therefore, it's likely to be easily acquirable. If it's easily acquirable, then there's no reason to hire for it.

- **Proxy Knowledge**: This is knowledge that is not quite at the specialist level (in fact, you might not even need it), but that you would expect a candidate at their level to know. For example, it might not be very important to you if a candidate knows CSS or HTML. But if a candidate has worked in depth with these technologies and can't talk about why tables are or aren't good, that suggests an issue. They're not absorbing information core to their job.

Academics use big 0, big 0 (theta), and big O (omega) to describe runtimes

- **O (big 0)**: In academia, big O describes an upper bound on the time. An algorithm that prints all the values in an array could be described as O(N), but it could also be described as O(N2), O(N3), or 0(2N) (or many other big O times). The algorithm is at least as fast as each of these; therefore they are upper bounds on the runtime. This is similar to a less-than-or-equal-to relationship. If Bob is X years old (I'll assume no one lives past age 130), then you could say $X \leq 130$. It would also be correct to say that $X \leq 1,000$ or $X \leq 1,000,000$. It's technically true (although not terribly useful). Likewise, a simple algorithm to print the values in an array is O(N) as well as O(N3) or any runtime bigger than O(N).
- Ω **(big omega)**: In academia, Ω is the equivalent concept but for lower bound. Printing the values in an array is $\Omega(N)$ as well as Ω(log N) and Ω(1). After all, you know that it won't be faster than those runtimes.
- Θ **(big theta)**: In academia, Θ means both O and Ω. That is, an algorithm is Θ(N) if it is both O(N) and Ω(N). Θ gives a tight bound on runtime.

In industry (and therefore in interviews), people seem to have merged Θ and O together. Industry's meaning of big O is closer to what academics mean by Θ, in that it would be seen as incorrect to describe printing an array as O(N2). Industry would just say this is O(N).

What is the relationship between best/worst/expected case and big 0/theta/omega? It's easy for candidates to muddle these concepts (probably because both have some concepts of "higher':"lower" and "exactly right"), but there is no particular relationship between the concepts.
Best, worst, and expected cases describe the big O (or big theta) time for particular inputs or scenarios. Big 0, big omega, and big theta describe the upper, lower, and tight bounds for the runtime.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require 0(n) space. If we need a two-dimensional array of size nxn, this will require O(n^2) space.

Drop the Constants: It is very possible for O(N) code to run faster than 0(1) code for specific inputs. Big O just describes the rate of increase. For this reason, we drop the constants in runtime. An algorithm that one might have described as 0(2^n) is actually O(N).

Drop the Non-Dominant Terms
What do you do about an expression such as $O(N^2 + N)$? That second N isn't exactly a constant. But it's not especially important. We already said that we drop constants. Therefore, $O(N^2 + N)$ would be O (N2). If we don't care about that latter N 2 term, why would we care about N? We don't.
You should drop the non-dominant terms:

- $O(N^2 + N)$ becomes $O(N^2)$.
- $O(N + logN)$ becomes $O(N)$.
- $O(5 * 2^N + 1000N^100)$ becomes $0(2^N)$.

We might still have a sum in a runtime. For example, the expression $0(B^2 + A)$ cannot be reduced (without some special knowledge of A and B).

- If your algorithm is in the form "do this, then, when you're all done, do that"then you add the runtimes.
- If your algorithm is in the form "do this for each time you do that"then you multiply the runtimes.

How do you describe the runtime of insertion? This is a tricky question.
The array could be full. If the array contains N elements, then inserting a new element will take O(N) time. You will have to create a new array of size 2N and then copy N elements over. This insertion will take O(N) time. However, we also know that this doesn't happen very often. The vast majority of the time insertion will be in O(l) time.

Log N Runtimes
We commonly see O(log N) in runtimes. Where does this come from? What is k in the expression $2^k = N$? This is exactly what log expresses.
$2^4 = 16$ $log_2l6 = 4$
$log_2N = k$ $2^k = N$

This is a good takeaway for you to have. When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a 0(log N) runtime.