Acing the Amazon SDE Interview

Sample Interview Topics
Below is a list of broad areas that we expect people to be familiar with. It is certainly not required that you memorize all of the information outlined below, but this should serve as a helpful reference guide for the types of things you might want to brush up on before interviewing with Amazon.

Programming Languages
We do not require that you know any specific language before interviewing for a technical position at Amazon.com, but familiarity with a prominent object-oriented language is generally a prerequisite for success. Not only should you be familiar with the syntax of a language like C++, Java, or C#, but you should also know some of the language's nuances such as how memory management works, what some of the most commonly used collections or libraries are, etc. You should be able to compare languages and talk about the tradeoffs between using language X vs. language Y. Additionally, it is considered a plus to be familiar with some scripting language such as perl, ruby, awk, etc. It is also nice to know the basics of regular expression as it is now a mainstay in both the object oriented and scripting worlds.

Data Structures
Most of the work we do involves storing and providing access to data in efficient ways. This necessitates a very strong background in standard data structures. You should know what each of these data structures is and how they are implemented, what their runtimes are for common operations, and under what circumstances it would be beneficial to use a particular one. The list below is in no particular order:

- Array
- Linked List
- Tree (Tree, Binary Tree, Binary Search Tree, Red-Black Tree, etc.)
- Heap
- Hash Table
- Stack
- Queue
- Tree
- Graph (both directed and undirected)
- Algorithms

It is also important to know efficient ways to manipulate data. One great way of demonstrating this is to brush up on some common algorithms. We will expect that you can apply and discuss the tradeoffs between commonly used algorithms.

- Sorting: Bubble Sort; Merge Sort; Quick Sort; Radix/Bucket Sort
- Traversals (On multiple data structures): Depth First Search; Breadth First Search

Coding
Expect to be asked to code syntactically correct code– no pseudo code. If you are a bit rusty coding without an IDE or coding in a specific language, it is probably a good idea to dust off the cobwebs and get comfortable coding with pen and paper. The most important thing a software engineer does at Amazon.com is write scalable, stable, robust, and well tested code. These are going to be the main criteria by which your code will be evaluated, so make sure that you check for edge cases and common error inputs, as well as the “happy paths” through the code.

Object Oriented Design
Good design is paramount to extensible, bug free, and long living code. It is possible to solve software problems in almost limitless numbers of ways; but when software needs to be robust and extensible,
It is important to know some common techniques that help with this. Using object oriented design best practices is one way to build lasting software. You should have a working knowledge of a few common and useful design patterns (singleton, factory, adapter, bridge, visitor, command, proxy, observer, etc.), as well as know how to write software in an object oriented way with appropriate use of inheritance and aggregation.

Databases
Most of the software that we write is backed by a database somewhere. A lot of the challenges we face come into play when interfacing with existing data models and when designing new data models. You should know the basics of how relational databases work, how to design relational database schemas, and how to write basic SQL queries against a database.

Distributed Computing
Our systems at Amazon.com usually have to work under very strict tolerances at high load. While we have some internal tools that help us with scaling, it is important to have an understanding of a few basic distributed computing concepts. Having an understanding of topics such as map-reduce, service oriented architectures, distributed caching, and load balancing, will help you in formulating answers to some of the more complicated distributed architecture questions you might encounter.

Internet Topics
This is Amazon.com; we are an online company and we expect our engineers to be at least familiar with the basics of how the internet works. You might want to brush up on how internet browsers do what they do, DNS lookups, what TCP/IP and HTTP are, sockets, etc. We are not necessarily looking for network engineering types of qualifications, but a solid understanding of the fundamentals of how the web works is a requirement.

Operating Systems
You will not need to know how to build your own operating system, but you should be familiar with some OS topics that can affect code performance; things such as memory management, processes, threads, synchronization, paging, multithreading, deadlocks (causes, detection, avoidance).

<u>**There will be 3 coding sessions:**</u>

1. ***Problem Solving***
    a. Did you correctly solve the problem?
    b. Did you reach an optimal solution?
    c. Can you justify decisions by identifying potential tradeoffs with several different solutions?
    d. Do you understand why one solution is better than another?
    e. Did you define the problem by asking relevant clarifying questions?

   f. Did you consider additional factors beyond the basic problem like developer effort, or team composition?
   g. Do you require minimal hints and able to use the hints effectively?
   h. How do you consider edge cases or difficult changes to the requirements?
   i. How do you react when your assumptions are challenged?
2. ***Logical & Maintainable Code***
   a. Is your code simple (leverages reuse, properly formatted, no improper coding constructs)?
   b. Is your code maintainable? (quickly able to trace impact of changes, clear variable naming conventions)?
   c. Is your code clear and understandable?
   d. Is your code following proper and consistent syntax?
   e. Does your code work as intended?
3. ***Data Structures & Algorithms***
   a. Do you know how to apply proper data structures & algorithms to solve a technical problem
   b. Do you know the tradeoffs of different data structures (ex. hash map vs tree vs array)
   c. Do you know the tradeoffs of different algos (breadth search vs depth search)
   d. Can you justify why the selected data structure and algorithm was used?
   e. Do you know how to translate these into a workable solution?
   f. Do you know the run time and space complexity tradeoffs of the different solutions?
   g. Can you optimize the solution?

<u>**General tips for doing well on the coding questions:**</u>

Before Coding:
1. Ask clarifying questions
2. Voice your assumptions – this will help the interviewer steer you back on track if you’re making fundamentally wrong assumptions before you start coding
3. Ask for examples and use them to validate your ideas

During Coding:
1. Explain your decision making
2. Describe your code
3. Write syntactically correct real code NEVER pseudocode. Try to be consistent in your naming conventions and it will make testing and optimizing easier later.
4. If you forget a specific API or specific link into an array or something like that, just ask the interviewer who should help you

After Coding:
1. Test your code
2. Talk through optimizing the solution if it can be further optimized if there is more time

Most important here is that you should focus on having a working and fully coded solution on the board to the problem by the end of each interview. If it’s easier for you to start with an inefficient brute force approach and then optimize from there at the end, it’s a perfectly valid approach. Just state that up front with your interviewer. Brute force solution is better than an incomplete solution.

The 4th technical session is System Design:

1. Ask clarifying questions up front to scope out the requirements of the system
2. Ask clarifying questions up front to scope out the requirements of the system
3. Draw it out on the board using boxes and use shapes to represent different components. Database, UI, etc. Use lines showing the relationships and flow between the different components
4. Keep things simple but go all the way down to the API level and into specific technology choices for the system you're designing.
5. Design for operational performance, have plans for failures, and be able to measure results (metrics).
6. Be prepared to identify potential shortcomings and discuss tradeoffs. Why are you choosing a specific technology or design? Discuss what other options you could use and what circumstances they might perform better under.
7. Be prepared to discuss scale. What happens if we go from 10,000 users to 1m users and how do we need to change our system to be performant?

Whiteboarding during your interview:

1. You will likely be asked to whiteboard during your interview (especially during the systems design portion)
2. Write a list of requirements on the board, and keep asking questions. This should be the first thing you write out.
3. Be prepared to whiteboard. Practice writing code and system design out by hand. Consider a scalable code or design before you begin drafting.
4. Keep the customer front of mind. Who is the customer, and what problem are you solving for them?
5. Dig for clarification. Your interviewer will not try to trick you. Questions may be intentionally vague to push your innovation.
6. Begin drawing a diagram once you've done enough digging to begin white boarding your system design solution. Start with shapes to represent different software components and data sources, and then arrows connecting them to show web services, APIs and interactions between components.
7. Know how your solution solves the problem. If you suggest technology to help solve, understand how that technology works.
8. Think out loud as you write out your code or system design.
9. Scaling is a critical component of software design. Consider software components, how to store data, APIs, component relationships, and data flows. Keep this in mind when diagramming and designing your software systems.
10. Be familiar with Distributed Systems, SOA, and n-tiered software architecture. This will help you answer system design questions.
11. Operational performance of your design is important. How will you ensure this system is working at an acceptable level of performance? If a problem occurs, what will be involved to trouble shoot and resolve quickly? What are the possible points of failure and how can they be made more robust against failure?
12. Interact with your interviewer. You will be asked several questions related to design. Engage with your interviewer with necessary questions to complete the exercise.

Tips from an Amazon SDM

There are a few aspects of your technical abilities that interviewers try to assess:

1. Problem solving - given an ambiguous technical problem, are you able to break it down, ask clarifying questions (yes - this is important and very easy to forget), come-up with multiple different approaches, talk about the trade-offs of time and space complexity and implement the solution for the most optimal solution (or the one you and your interviewer agree to) by writing code.
2. Writing logical and maintainable code - Amazon is always evolving based on our customer's needs. So is our software and processes. Having to re-write the code of a feature in a microservice every single time our customer's needs have changed or there are new use-cases is not feasible. We build code which is extensible and can very easily adapt to the growth needs of our customers and business. There will be one interview where your ability to write code which is well structured, logical, extensible and maintainable is going to be evaluated.
3. Data structures and algorithms - Every single technical problem that we solve at Amazon, although it may sound straight-forward and right out of a Computer Science text book, is very challenging and difficult to solve. The simple reason being - Amazon's scale. Once you add the aspect of scale (i.e. several million transactions per second, several thousand of servers that handle customer's requests, ultra-low-latency systems that respond to requests from IoT devices etc.), all these straight-forward computer science problems become extremely complex to solve. This is where being well versed and having an in-depth understanding of data structures and algorithms is extremely important. We write code that uses the most optimal data structures and we optimize for compute and memory performance. One of the interviews is going to focus on evaluating your ability to use data structures and algorithms to solve the problem posed by your interviewer.
4. System design - Very similar to the challenges described in the previous section, at Amazon, we deal with significant scale challenges and we anticipate the needs of our customer and innovate on their behalf. This requires us to design and build systems that scale in a virtually unlimited fashion, integrate across different platforms/ecosystems and meet our customer where they are (i.e. support for multiple platforms/languages/frameworks etc.). One of your interviews will be focused on evaluating your ability to design and draw/represent a system for a hypothetical but very similar to real-world scenario.

Answering Technical Interview Questions

1. Never assume anything. **Ask clarifying questions**. Why? Here is why - imagine your friend comes up to you and asks - "Hey, can you please buy me some bread?". Would you head out and buy a loaf of sourdough bread and hand it to your friend? Highly unlikely. You would ask your friend a few questions - "*What type of bread? Regular or gluten-free? Price range? How many loafs do you want? Any particular store of choice? When do you need it? What if that bread isn't available, can I buy something else instead? What "best if used before" date would you like me to buy?"* We always deal with a lot of ambiguity. So, it's important to disambiguate to the furthest extent possible and assume only after confirming with your interviewer.
2. Never jump into writing code as soon as you hear the problem statement. First think about how you would solve the problem.
3. Think out loud. If you are thinking of possible solutions, write them down on the whiteboard or live coding session.
4. Think of multiple approaches. Sometimes what you consider to be the most optimal may not be the one that was expected because you made a few assumptions about the problem statement.

5. Talk about space/time trade off of the solutions you come-up with.
6. Work with your interviewer and pick a solution/approach before you start writing code.
7. It is important to keep track of the time remaining. Your interviewer does keep the time in mind but try to be cognizant of the amount of time you are spending on different aspects. Ensure that you keep enough time for you to write code. Practice writing on a sheet of paper/whiteboard or for phone interviews, practice typing.
8. After you are done coding, TEST! Write down failure cases, talk your interviewer through scenarios where your solution might fail and talk through potential ways to address those scenarios. Think Amazon scale.
9. Think like a customer of the solution you've implemented. Critique your approach and try to see what can be improved.
10. Generally, SDE roles do not require you to write code in any specific programming language but be sure to check with your interviewer.
11. Don't always assume that you can use built-in libraries of a particular language to perform - sort/search/graph search/tree traversal etc. Check with your interviewer if it is ok to use or if you are expected to implement that functionality.
12. Don't prematurely try to optimize your solution. i.e. spending several minutes on trying to reduce the number of lines of code or optimizing number of temp variables etc. to perform an operation before you have a fully working end to end solution. First and foremost, get a working solution and then look for opportunities for optimization.
13. Use meaningful variable/method/class naming conventions and use general coding best practices.
14. For system design
    a. Be sure to gather all the requirements before proposing any solutions.
    b. Don't just draw magical black-boxes in your system/architecture diagram. Be very specific as to what each box is and have an example of what technology you would use. Talk about trade-offs between using a relational DB versus a key-value store, trade-offs in different compute techniques - hosted VMs (EC2) vs. containers vs. Lamdas etc.
    c. Think about ALL aspects of building a system - scale, security, extensibility, localization/internationalization, monitoring, logging, auditing etc.
15. Lastly, if writing on the whiteboard, make sure the handwriting is legible.