BITMAKER LABS

HIRING HANDBOOK

Section I: Problem Solving

In this section we start laying the foundation for solving the technical questions that you might be asked during an interview. For all of our interview prep, there's a right way and a wrong way of progressing through it. The wrong way is to read through the problem and its solution, the right way is to actually practice solving it.

Here's how we suggest that you practice solving these problems:

1. Solve the problem on your own
2. Write the code for the algorithm on paper
3. Test your code (or solution) on paper
4. Type your written code as-is into a computer
5. List all of the errors that you made so you know where to improve

When you're actually in an interview setting, you can apply a five-step approach to solving problems:

1. Ask your interviewer questions to resolve ambiguity
2. Design an algorithm
3. Write pseudocode first
4. Write your code at a moderate pace
5. Test your code and carefully fix any mistakes

Step 1: Resolve Ambiguity

It's important to know all of the information that's relevant to solving a problem before you attempt it. Start a dialogue with your interviewer by asking questions that will help you solve the problem. Many interviewers will assess the caliber of the questions that you ask, so doing this is very important.

The questions that you ask should be like the following: What are the data types? How much data is there? What assumptions do you need to solve the problem? Who is the user?

Example: "Design an algorithm to sort a list."

Question: What sort of list?
Answer: An array.

Question: What does the array hold? Numbers? Strings? Characters?
Answer: Numbers.

Question: And are the numbers integers?
Answer: Yes.

Question: Where did the numbers come from? Are they IDs? Values of some field?
Answer: They are customer ages.

Question: And how many customers are there?
Answer: Around a million.

You now know much more about your problem: sort an array containing a million integers between 0 and 130 (a max age). How do we solve this? By creating an array with 130 elements and counting the number of ages at each value.

Step 2: Create an Algorithm

We recommend that you first mention a possible brute force solution, that is, a solution that is neither elegant nor efficient, but will get the job done given enough time and resources. You can then optimize your answer from there. When you're trying to develop a more optimal algorithm keep the following questions in mind:

- What are its space and time complexity?
- What happens if there is a lot of data?
- Does your design cause other issues?
- If there are other issues or limitations, did you make the right trade-offs? What scenarios would make the trade-offs less optimal?
- If you're given specific data (e.g. data is ages, or sorted) have you leveraged that information? It was probably given to you for a reason.

Step 3: Write Pseudocode

By writing pseudocode first you can outline your thoughts clearly and reduce the number of mistakes you commit when writing real code. Make sure to tell your interviewer that you're writing pseudocode first and that you'll follow with the real deal.

Step 4: Code

Go at a slow and methodical pace and keep these two principles in mind:

- Use data structures generously: Create a good data structure or define your own. E.g. if asked a problem involving finding a minimum age of a group of people, consider defining a data structure to represent a Person. This shows that you care about object-oriented design.

- Don't crowd your coding: When writing code on a whiteboard, start in the upper left hand corner rather than in the middle.

Step 5: Test

Make sure to test your code by walking through it after completion. If your algorithm is complicated consider testing while you write the code instead of after. When you find mistakes, carefully think through why the bug is occurring before fixing the mistake. Consider testing for extreme cases, user error and general cases.

Algorithmic Problems

Solving algorithm-based questions can be difficult, but the more you practice the easier it gets. There are multiple approaches to solving these problems and the more problems you solve the easier it is to determine the approach that you should use for each type of question. Our four favorite approaches are as follows:

Approach I: Example and Expand

Under this method, you write out specific examples of the problem and see if you can derive a general rule from there.

Example: Given a time (3:27), calculate the angle between the hour and minute hands on a clock.

We suggest you start by drawing a picture of a clock stopped at this time. For the below solution, we'll assume that the hour hand is `h` and the minute hand is `m`. we'll also assume that the hour specified is an integer between 0 and 23 inclusive.

By playing around with these examples, we can develop a rule:

- Angle between the minute hand and 12 o'clock: `360 * m / 60`
- Angle between the hour hand and 12 o'clock:
  `360 * (h % 12) / 12 + 360 * (m / 60) * (1 / 12)`
- Angle between hour and minute: `(hour angle – minute angle) % 360`

By reducing this, we get the answer: `(30h - 5.5m) % 360`

Approach II: Pattern Matching

Under this approach, we consider what problems the algorithm is similar to and try to modify the solution so it applies for this problem.

Example: A sorted array has been rotated so that the elements might appear in the order 3 4 5 6 7 1 2. How would you find the minimum element? You may assume that the array has all unique elements.

Two problems that come to mind as similar are:

- Find the minimum element in an array
- Find a particular element in a sorted array (i.e., binary search)

You'll quickly realize that finding the minimum element by iterating through each position isn't a very elegant solution. But using binary search is. You know that the array is sorted, but rotated. So, it must proceed in an increasing order, then reset, and increase again. The minimum element is the reset point.

If you compare the middle and last element (6 and 2), you will know the reset point must be between those values, since `MID > RIGHT`. This wouldn't be possible unless the array was reset between those values.

If `MID` were less than `RIGHT`, then either the reset point is on the left half, or there is no reset point (the array is truly sorted). Either way, the minimum element could be found there.

We can continue to apply this approach, dividing the array in half in a manner much like binary search. Eventually we'll find the minimum element.

Approach III: Simplify and Generalize

With this approach, we implement a multi-step process. First, we change a constraint such as the data type or amount of data. Doing this helps us simplify the problem. Second, we solve this simplified version of the problem. At last, once we have an algorithm for the simplified problem, we generalize the problem and try to adapt the earlier solution for the more complex version.

Example: A ransom note can be formed by cutting words out of a magazine to form a new sentence. How would you figure out if a ransom note (represented as a string) can be formed from a given magazine (string)?

To simplify the problem, we can modify it so that we are cutting characters out of a magazine instead of whole words.

We can solve the simplified ransom note problem with characters by simply creating an array and counting the characters. Each spot in the array corresponds to one letter.

First, we count the number of times each character in the ransom note appears and then we go through the magazine to see if we have all of those characters.

When we generalize the algorithm, we do a very similar thing. This time, rather than creating an array with character counts; we create a hash table that maps from a word to its frequency.

Approach IV: Build from a Base Case

For this approach we solve the problem first for a base case (e.g., `n = 1`). This usually means just recording the correct result. Then, we try to solve the problem for `n = 2`, assuming that you have the answer for `n = 1`. Next, we try to solve it for `n = 3`, assuming that you have the answer for `n = 1` and `n = 2`.

We can eventually build a solution that can always compute the result for `N` if we know the correct result for `N - 1`. It may not be until `N` equals 3 or 4 that we get an instance that's interesting enough to try to build the solution based on the previous result.

Example: Design an algorithm to print all permutations of a string. For simplicity, assume all characters are unique.

Consider a test string "abcdefg":

```
Case "a" --> {"a"}
Case "ab" --> {"ab", "ba"}
Case "abc" --> ?
```

This is the first case that will lead us to discovering the pattern. If we had the answer to `P("ab")`, how could we generate `P("abc")`? Well, the additional letter is `"c"`, so we can just add `"c"` in at every possible position. That is:

```
P("abc")= insert "c" into all locations of all strings in P("ab")
P("abc")= insert "c" into all locations of all strings in
{"ab","ba"}
P("abc")= merge({"cab", "acb", "abc"}, {"cba", "bca", bac"})
P("abc")= {"cab", "acb", "abc", "cba", "bca", bac"}
```

Now that we understand the pattern, we can develop a general algorithm that happens to be recursive. We can generate all permutations of a string `s1…s(n)` by removing the last character and generating all permutations of `s1…s(n-1)`. Once we have the list of all permutations of `s1…s(n-1)` we iterate through this list, and for each string in it, we insert `s(n)` into every location of the string. This type of approach often leads to naturally recursive algorithms.

Brain Teasers:

Companies incorporate brainteasers as part of their interview process to see how you walk through a problem and if you attempt to solve it in a logical, procedural fashion. Answer these questions out loud, you might find it helpful to start talking through the problem with the interviewer shortly after receiving it. Getting these questions correct isn't what an interviewer is concerned with; the interviewer is far more concerned with evaluating your ability to problem-solve. The first step to solving these problems is to develop rules and find patterns.

Example: You have two wooden planks, and each takes one hour to burn. How would you use them to time exactly 15 minutes? Note that the thickness of each plank is not uniform, that is, half the plank does not necessarily take half an hour to burn.

What we know:

- We can time one hour by burning one plank.
- We can time two hours, by lighting one plank, waiting until it burns out and then lighting the second. We can generalize this into a formula.

Rule 1: Given a plank that takes x minutes to burn and another that takes y minutes, we can time x+y minutes.

Another thing to consider is that there's more than one way to burn a plank. You could burn a plank by igniting one end, igniting the middle or igniting both ends. Igniting one end led us to our first rule, igniting the middle isn't very useful, but igniting both ends leads us to our second rule. Burning a plank from both ends should take half the time.

Rule 2: Given a plank that takes x minutes to burn, we can time x/2 minutes by lighting the plank at both ends.

What we know:

- We can time 30 minutes using one plank.
- We can remove 30 minutes of burning time from the second plank by lighting plank 1 on both ends and plank 2 on just one end starting at the same time.

Rule 3: If plank 1 takes x minutes to burn and plank 2 takes y minutes, we can turn plank 2 into a plank that takes (y - x) minutes or (y – x/2) minutes.

What we know:

- We can turn plank 2 into a plank with 30 minutes of burn time.
- If we then light plank 2 on the other end (see rule 2), plank 2 will be burned after 15 minutes.

We now know enough to solve this problem. Based on what we've discovered, here's the solution:

1. Light plank 1 at both ends and plank 2 at one end.
2. When the two flames on plank 1 meet, 30 minutes will have passed. Plank 2 has 30 minutes left of burn-time.
3. At that point, light plank 2 at the other end.
4. In exactly 15 minutes, plank 2 will be completely burnt.

Another common type of brainteaser problem involves minimizing worst-case scenarios. These questions involve doing something the least amount of times, or a predetermined number of times. A useful way of solving these problems is to "balance out" the worst case. That is, if an early decision results in a skewing of the worst case, we can sometimes change the decision to balance out the worst case to arrive at an optimal conclusion.

Example: You have nine balls. Eight are the same weight and one is heavier. You are given a scale, which tells you only whether the left side or the right side is heavier. Find the heavy ball in just two uses of the scale.

Here's one possible approach: Divide the balls in sets of four, with the ninth ball sitting off to the side. The heavy ball is in the heavier set. If they are the same weight, then we know that the ninth ball is the heavy one. Replicating this approach for the remaining sets of four would result in a worst case of three iterations, or one too many.

This is an example of the worst case: the ninth ball takes just one weighing to discover if it's heavy, whereas others take three. If we penalize the ninth ball by putting more balls off to the side, we can lighten the load on the others. This is how to balance a worst case.

If we divide the balls into sets of three items each, we will know after just one weighing which set has the heavy one. We can even formalize this into a rule:

Rule: Given N balls, where N is divisible by 3, one use of the scale will point us to a set of N/3 balls with the heavy ball.

For the final set of three balls, we simply repeat this: put one ball off to the side and weigh two. Pick the heavier of the two. Or, if the balls are the same weight, pick the third one.

If neither of these suggestions works, consider using one of the four algorithmic approaches. Example and expand, simplify and generalize, and build from base case are particularly useful.

Brain Teaser Interview Questions

1. You have 20 bottles of pills. 19 bottles have 1.0-gram pills, but one has pills of weight 1.1 grams. Given a scale that provides an exact measurement, how would you find the heavy bottle? You can only use the scale once.

2. There is an 8x8 chessboard in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer by providing an example or showing why it's impossible.

3. You have a five-quart jug, a three-quart jug, and an unlimited supply of water (but no measuring cups). How would you come up with exactly four quarts of water? Note that the jugs are oddly shaped, such that filling up exactly 0.5 of a jug would be impossible.

4. There's a bunch of people living on an island, a visitor comes with a strange order: all blue-eyed people must leave the island as soon as possible. There will be a flight out at 8:00pm every evening. Each person can see everyone else's eye color, but they do not know his or her own and no one is allowed to ask. Additionally, they do not know how many people have blue eyes, although they do know that at least one person does. How many days will it take the blue-eyed people to leave?

5. There is a building of 100 floors. If an egg drops from the Nth floor or above, it will break. If it's dropped from any floor below, it will not break. You're given two eggs. Find N, while minimizing the number of drops for the worst case.

6. There are 100 closed lockers in a hallway. A man begins by opening all 100 lockers. Next, he closes every second locker. Then, on his third pass, he toggles every third locker (closes it if it is open or opens it if it is closed). This process continues for 100 passes, such that on each pass `i`, the man toggles every `i`th locker. After his 100th pass in the hallway, in which he toggles only locker #100, how many lockers are open?

Section II: Behavioural

In this section we talk about the "softer" skills required to do well in an interview setting. These questions are often asked in a manner that reveals your personality while an interviewer attempts to better understand your past experience and your potential fit within the organization.

Interviewers often ask questions that begin with "Tell me a time when you…" and expect an answer that's derived from specific past work or project experiences. Preparing to answer these questions is easiest by filling up a grid similar to the example below.

Common Questions	**Project I**	**Project II**	**Project III**	**Project IV**
Most Challenging				
What You Learned				
Most Interesting				
Hardest Bug				
Hardest to Learn				
Conflicts Resolved				

Along the top columns you should list all the major aspects of your resume, including each project, job or activity. In the case of your experiences at Bitmaker, choose the projects that had the most impact on you – it's easier to tell a story about something that you're passionate about than something you're not, we also suggest using some of the more complex projects instead of the simpler ones. Along the side rows you should list the common questions an interviewer might ask, like the above examples. In each cell, put a corresponding story.

Although it's good to plan, avoid making the response sound scripted. The easiest way to do this is to keep the story condensed to just a few key words that help you remember the point that you're trying to get across. Remember that when answering questions you're not just looking for an answer, you're looking for an answer that reveals something about yourself.

What to Ask Your Interviewer

To reiterate, interviewers will judge you on the questions that you ask, so make sure to ask questions that demonstrate your knowledge or show your interest in their company. Specific questions are practical in nature and will allow you to ask questions about what it's like working for the company you're interviewing with. Insightful questions are used to show that you've done your homework on the company before interviewing, in addition to demonstrating your knowledge of different technologies or best practices. Enjoyment questions are used to show that you're keen to learn new things.

Specific Questions

1. "How much of your day to you spend coding?"
2. "How many hours do you spend in meetings every week?"
3. "What is the ratio of developers to sales people in your organization?"

Insightful Questions

1. "I noticed that you use both Rails and Django in your technology stack. What advantages are there to using both? What are the drawbacks?"
2. "I noticed that you use PostgreSQL instead of MySQL. What led you to make that choice?"

Enjoyment Questions

1. "I didn't get much of a chance to use NoSQL databases at Bitmaker, but I'm very interested in this. I noticed that you're an expert on the topic, I've self-taught myself a bit about it but want to learn more. What are your suggestions on how I can move from a "novice" stage to an "intermediate" one?
2. "I'm very interested in scalability. Did you come in with a background in this, if not, what kind of opportunities did you have to learn about it while working here?"

Best Practices for Answering Behvaioural Questions

1. Be specific, not arrogant. State what you did, and if your contribution was the most important part of a project be sure not to sound boastful.
2. Limit the details of your response. Keep your answers to the point and don't talk about the minute details of the problem you encountered. Interviewers who don't understand the nuances of the story you're telling will tune you out if you're too long-winded.
3. Give structured answers. Lead with a "headline" that summarizes the story e.g. "Let me tell you about a time where I controlled a robot with a Dance Dance Revolution mat. I started by…"

4. Use the C.A.R. (context, action, result) approach to storytelling. Give your interviewer a brief overview of the situation that you're in. Progress into the actions you took to resolve your problem or improve your circumstances. Finish with the result or the outcome of the actions that you took. The context and the result should be kept short; interviewers care more about the actions that you took.

Building Your Network

Many people get jobs through their immediate connections, but an even larger number of people get jobs through their secondary, or even tertiary connections. It's critical to build a good network so you can have an easier time finding out about jobs. A good network has reach while remaining genuine. A genuine network is filled with people that will actually assist you if they were asked. A network with reach should cover multiple technology sub-industries and even extend beyond technology companies into traditional businesses with growing technology departments. Here are some tips that you can use to build a strong network:

1. Go to meetups, hackathons, speaker series and conferences.
2. Talk to people. Initiate conversation and introduce yourself, otherwise you'll never expand your network.
3. Be authentic and open about your interests and express an interest in the passions of other people. If they seem particularly interesting, ask to grab a coffee or beer so you can chat more.
4. Follow-up after meeting someone by adding him or her to LinkedIn.
5. Be helpful. By offering other people something now they're more likely to think of you highly and assist in the future when asked.

Writing Your Resume

A resume that's poorly written or poorly formatted is the easiest way to be excluded from an interview. Spend some time on this and make sure to highlight the skills and knowledge that you're the most interested in. We've found that the best practices for creating a resume are as follows:

1. Keep it to one page. Recruiters only spend a fixed amount of time looking at the resume of each candidate. By limiting your content to the most impressive things, your recruiter has a better chance of seeing them.
2. Write strong bullets to showcase your experience. Use the same C.A.R. method to show the context, action and result of your accomplishments.
3. Have an awesome project (or a few). Mention your project and the technologies that you used to build it.
4. List the programming languages that you know. Avoid talking about software; it doesn't matter to employers if you can use Word or Excel.

Engagement Strategies & Tools

Although we make it as convenient as possible for employers to contact and interview you, it would be conservative to assume that you will still have to work hard to receive a job offer at a company that interests you. In order to succeed you need to adopt the appropriate mindset for experiencing success while hunting for a job:

Be Resourceful

Finding a job as a web developer takes many of the same skills and thought processes that actually being a web developer requires. The tactics and strategies that you apply when troubleshooting a problem with your code are similar to the ones that you need to use when finding a job. Googling for potential employers and discussing opportunities with everyone you know will increase your likelihood of success.

An extremely useful tool that lets you be more resourceful is the Gmail extension Rapportive (http://rapportive.com/). Rapportive is a great tool for easily finding a person across multiple social networks (Twitter, LinkedIn, etc.). By using this you'll have more information on the people that you're interviewing or meeting with and be able to easily converse with them about their interests. Rapportive is also also a great tool for figuring out the correct email address for people that you're considering reaching out to. Generally, corporate email addresses conform to one of the following conventions:

- First_name@company.com
- First_nameLast_name@company.com
- First_initialLast_name@company.com
- First_initial@company.com
- First_initalLast_initial@company.com

By guessing at these with Rapportive installed, you'll be able to find a person's correct email address when Rapportive displays their related profiles.

Be Persistent & Proactive

Most people often forget to follow up after an interview or initial conversation. By doing this, they're letting a warm lead go "dead". Even if you don't forget, relying on an interviewer to follow up with you leads to you losing control of the conversation. Typically, after an interview with a potential employer it's a best practice to send them an email later that day. Thank them for the time they took to consider you for the position and let them know that you enjoyed the conversation and are interested in chatting again in the future. In the event that you interact with a prospective hirer in a more social setting, email them the next day asking to go out for coffee and to discuss what it's like working at their company.

In the case that you don't hear back after an initial email interaction with an employer don't assume that it means that they're not interested in hiring you. People are busy and often forget to respond to emails, so make sure to re-engage them after not hearing back from them within three days. The Gmail extension Boomerang (http://www.boomeranggmail.com/) is incredibly useful for setting reminders for yourself to follow up with people that you haven't heard back from. By using Boomerang, you can have messages reappear at the top of you inbox if they haven't been replied to over a certain period of time.

The final tool that we believe is absolutely essential to use when looking for employment is Yesware (http://www.yesware.com/). This is a very powerful add-on for either Chrome or Firefox. One incredibly useful feature that this tool offers is the ability to track which of your emails has been opened, where they were opened and how many times they've been opened. This will give you insight into how interested the person that you're contacting is to interact with you. Another powerful feature of Yesware is the ability to create templates for frequently repeated emails. Many initial contact emails and follow up emails are largely the same and have minor modifications regarding specific information – use this to your advantage by using templates for repetitive interactions.

Be Patient

Companies often take a long time to hire people. The interview process can take anywhere between two and five weeks and its very rare that an offer is made prior to this time period, especially with larger organizations. During this time it's important that you continue to source potential jobs to avoid becoming dependent on one slow-acting organization. Also, it's essential that you continue to code and continue to learn new, relevant topics.

While your waiting, we suggest blogging or tweeting about what you're learning and what you're building. This will help spread the awareness of your personal brand. Buffer (http://bufferapp.com/) is a great application that makes organizing the flow of your social media posts significantly easier.

Questions?

If you have any questions about the above strategies or tools, please don't hesitate to email Matt (matt@bitmakerlabs.com) to book a time to sit down to chat.

Section III: Technical

In this section we talk about the "softer" skills required to do well in an interview setting. These questions are often asked in a manner that reveals your personality while an interviewer attempts to better understand your past experience and your potential fit within the organization.

Recognizing Your Proficiencies

For a beginner, it's very difficult to determine your level of proficiency with a certain library or language. Short of having a veteran developer sit down to pair-program with you for half and hour to an hour it's hard to establish your level of understanding with each language or framework. Smarterer attempts to remedy this and we view its quizzes as a good tool to indicate areas for improvement. For a list of the quizzes we recommend that you complete please see the table below, tests with an asterisk are optional or more advanced in nature. We suggest that any test you take where you score less than "proficient" on is an area that you should attempt to improve.

Tests (*Optional)	URL(s)
Ruby	http://smarterer.com/tests/ruby
Ruby on Rails	http://smarterer.com/tests/ruby-on-rails
Git	http://smarterer.com/tests/git
HTML	http://smarterer.com/tests/html http://smarterer.com/tests/html5
CSS	http://smarterer.com/tests/css http://smarterer.com/tests/css3
JavaScript	http://smarterer.com/tests/javascript
jQuery	http://smarterer.com/tests/jquery
Software Engineering Concepts	http://smarterer.com/tests/software-engineering-concepts
Unix*	http://smarterer.com/tests/unix
Sublime Text*	http://smarterer.com/tests/sublime-text
SQL	http://smarterer.com/tests/sql
Design Patterns*	http://smarterer.com/tests/design-patterns
Data Structures and Algorithms*	http://smarterer.com/tests/data-structures-and-algorithms

Concepts, Algorithms & Data Structures

This section of the prep course covers many of the fundamental concepts and essential knowledge that is required of a computer programmer. Many of these topics exceed the scope of what an employer would expect of a web developer, but any additional knowledge that you possess in this area will make you more competitive with graduates of the leading computer science programs.

Big-O Notation

Programmers will often have discussions about the efficiency at which their code is able to solve problems. Big-O notation is a way of easily explaining this efficiency. An algorithm or a function (we can treat these the same for efficiency sake) within your program has a Big-O notation that's determined by how it responds to different volumes of inputs. Consider the following code:

```
def item_in_list(to_check, the_list)
  the_list.each do | i |
    if to_check == i
      return true
    end
  end
end
```

If we call this function like `item_in_list(2, [1,2,3])`, it should finish executing quickly. The program loops over each thing in the list and if it finds the first argument to our function, it returns True. If it doesn't find the first argument it prints out all of the numbers in the second argument. The complexity of this function is O(n), or "Order of N". What this means is that the time it takes to run this function is linearly correlated with the number of the items in its input array.

If we were to graph this on a Time vs. Input axis, the line would be drawn as a straight line that points up and to the right. It's important to note that the above code will take different times to complete depending on the items position within the array, with the worst case being that the item is not found in the array. O(n) measures the worst case scenario. Here's another example:

```
def return_item(item)
  return item
end
```

This function is called O(1), which is known as "constant time". What this means is that regardless of the size of our inputs, the computation will always take the same amount of time. This is considered the best case for any function. As a final example, see the code below:

```
def all_combinations(the_list)
  results = []
    the_list.each do |item|
      the_list.each do |inner_item|
        results << item << inner_item
      end
    end
    return results
end
```

This function matches every item in the list with every other item in the list. If we gave it an array `[1,2,3]` we'd get back `[(1,1) (1,2), (1,3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]`. This function (or algorithm) is considered O(n^2). This is because for every item in the list (assume input size of n), we have to do n more operations. So n * n == n^2. If this were drawn out, the line would look curved, as the time that it takes to execute increases exponentially with the number of inputs. To summarize: the more loops within loops a program has, the longer it takes to process all of its inputs.

Object-Oriented Design

These questions require a candidate to sketch out the classes and methods to implement technical problems or real-life objects. The reason for asking these questions isn't for you to regurgitate design patterns, but to show your ability to create elegant, maintainable object-oriented code. It's vital to correctly demonstrate your knowledge on these types of questions. By following the below steps, you'll be more likely to resolve at an answer that the interviewer approves of.

Step 1: Resolve Ambiguity

Object-oriented design (OOD) questions are intentionally vague – interviewers want to assess whether or not you'll make assumptions or ask clarifying questions. It's critically important that you demonstrate your willingness to form a complete understanding of the purpose of the software you're developing. Otherwise, you'll be wasting your employers' time and money solving a problem using incorrect assumptions.

When being asked this type of question, you should inquire who is going to use it and how they are going to use it. You may want to go as far as to go through who, what, where, when, how, and why.

Example: Describe the object-oriented design for a coffee maker.

This coffee maker might be an industrial machine that's designed to be used in a massive restaurant servicing thousands of customers per hour and making one hundred different kinds of coffee products. On the flip side, it could be a simple, single serving machine for household use. The type of this machine significantly alters your design.

Step 2: Define the Core Objects

Now that we understand the purpose of what we're designing, we should consider what the possible "core objects" in our system are. For example, suppose we are asked to do the object-oriented design for a restaurant. Our core objects might be things like `Table`, `Guest`, `Party`, `Order`, `Meal`, `Employee`, `Server`, and `Host`.

Step 3: Analyze Relationships

Having more or less decided on our core objects, we now want to analyze the relationships between the objects. Which objects are members of which other objects? Do any objects inherit from any others? Are relationships many-to-many or one-to-many?

Example: The restaurant question above may end up with us using the following design

- `Party` should have an array of `Guests`.
- `Server` and `Host` inherit from `Employee`.
- Each `Table` has one `Party`, but each `Party` may have multiple `Tables`.
- There is one `Host` of the `Restaurant`.

There are multiple "gotchas" that you should watch out for, can't a single `Table` have multiple `Parties`?

Step 4: Investigate Actions

You should now have the basic outline of your object-oriented design. The only thing that remains is to consider the key actions that the objects will take and how they relate to each other. This may lead you to the realization that you may have forgotten some objects, and you will need to update your design.

Example: a `Party` walks into a `Restaurant`, and a `Guest` makes a request for a `Table` from the `Host`. The `Host` looks up the `Reservation` and, if it exists, assigns the `Party` to a `Table`. Otherwise, the `Party` is added to the end of the list. When a `Party` leaves, the `Table` is freed and assigned to a new `Party` in the list.

Design Patterns

Most interviewers won't ask you questions related to the specific types of design patterns you use to program, but you should be familiar with what they are. Two of the most frequently used types are featured below.

Singleton Pattern

The concept of this pattern is fairly straightforward: only a single instance of a class can exist. This is useful for when an application needs to ensure only one object is instantiated for each type of class. An example of this type of pattern is displayed below:

```
class MySingleton
  private_class_method :new
  @@instance = nil
  def self.instance()
    if (@@instance.nil?())
      @@instance = self.new()
    end
    return @@instance
  end
end
```

Factory Method Pattern

Using this pattern involves the creation of an object that builds other objects. It's best used when building a variety of objects of the same type, based on some input. Using this method should involve the creation of an object, rather than a static class. An example of this can be seen below.

```
class Report
  def get_printer(type)
    throw "Abstract: Please Overload"
  end
  def print_report(type)
    printer = get_printer(type)
    printer.print_header(@header)
    printer.print_body(@body)
    printer.print_footer(@footer)
  end
end
class DraftReport < Report
  def get_printer(type)
    if (type == "ascii")
      return AsciiDraftPrinter.new()
    elsif (type == "postscript")
      return PostScriptDraftPrinter.new()
    end
  end
end
class BasicReport < Report
  def get_printer(type)
    if (type == "ascii")
      return AsciiPrinter.new()
    elsif (type == "postscript")
      return PostScriptPrinter.new()
    end
  end
end
```

Object Oriented Design Interview Questions

1. Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.
2. Imagine you have a call center with three levels of employees: respondent, manager, and director. An incoming telephone call must be first allocated to a respondent who is free. If the respondent can't handle the call, he or she must escalate the call to a manager. If the manager is not free or not able to handle it, then the call should be escalated to a director. Design the classes and data structures for this problem. Implement a method `dispatchCall()` which assigns a call to the first available employee.
3. Design a musical jukebox using object-oriented principles.
4. Design a parking lot using object-oriented principles.
5. Design the data structures for an online book reader system.
6. Implement a jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle. You can assume that you have a `fitsWith` method which, when passed two puzzle pieces, returns true if the two pieces belong together.
7. Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?
8. Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves. The win is assigned to the person with the most pieces. Implement the object-oriented design for Othello.
9. Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

Strings and Arrays

Many interviewers will ask questions regarding popular data structures. From our experience, the primary type of data structure that our students receive questions on involve strings or arrays. We suggest going through the below questions to improve your ability to answer this type of question.

Strings and Arrays Interview Questions

1. Create an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?
2. Given two strings, write a method to decide if one is a permutation of the other.
3. Write a method to replace all spaces in a string with '%20'. For example, an input of "`Bitmaker Labs Summer 2013`" would produce an output of "`Bitmaker%20Labs%20Summer%202013`".
4. Implement a method to perform basic string compression using the counts of consecutive repeated characters. For example, the string "`abbccccaa`" would become "`a1b2c5a2`". If the compressed string isn't shorter than the original string then your method should return the original string.
5. Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?
6. Write an algorithm such that if an element in an MxN matrix is 0, its entire row and column are set to 0.
7. Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, s1 and s2, write code to check if s2 is a rotation of s1 using only one call to `isSubstring` (e.g. "waterbottle" is a rotation of "erbottlewat".

Sorting and Searching

By understanding common sorting and searching algorithms you'll have a better understanding of the efficiency of a selection of code.

Example: Given a large array of `Person` objects, sort the people in increasing order of age.

Given this question, we learn two very important pieces of information. First, it's a large array so efficiency is very important. Second, we're sorting based on ages, so we know that the values are in a relatively small range. If we were to scan through the various common sorting algorithms, we would notice that bucket sort might be an ideal option for this question. By making the buckets small (1 year each) we can get an O(n) running time.

Common Sorting Algorithms

Bubble Sort | Runtime: O(n^2)

Using bubble sort, we start at the beginning of an array and swap the first two elements if the first is greater than the second. We move on to the next pair, and then the next after that, all the while swapping numbers until the array is sorted.

```
class BubbleSort
  def sort(to_sort)
    sorted = false

    until sorted
      sorted = true

      for index in 0..(to_sort.length - 2)
        if to_sort[index] > to_sort[index + 1]
          sorted = false
          to_sort[index], to_sort[index + 1] = to_sort[index +1],
          to_sort[index]
        end
      end
    end

    return to_sort
    end
end
```

Selection Sort | Runtime: O(n^2)

Selection sort is another inefficient but simple sort. Its complexity is the same as bubble sort because of its nested loop. This sort occurs in-place, meaning it uses a small constant amount of data to perform its sort.

Selection sort works by dividing the array in two; the sorted array at the beginning and the unsorted array at the end. Initially, the whole array is unsorted. The algorithm iterates over the array finding the smallest element. It then swaps this element with the element at position 0. An example of this can be seen below.

```
class SelectionSort
  def sort(to_sort)
    for index in 0..(to_sort.length - 2)
      # select the first element as the temporary minimum
        index_of_minimum = index

        # iterate over all other elements to find the minimum
        for inner_index in index..(to_sort.length - 1)
          if to_sort[inner_index] < to_sort[index_of_minimum]
            index_of_minimum = inner_index
          end
        end

        if index_of_minimum != index
          to_sort[index], to_sort[index_of_minimum] =
          to_sort[index_of_minimum], to_sort[index]
        end
    end

    return to_sort
  end
end
```

Insertion Sort | Runtime: O(n^2)

Insertion sort has the same complexity as the previous two algorithms but is considered more efficient in practice than either of them. The way this works is by going through the elements to be sorted and inserting them in their correct order in a new sorted array. An example of this can be seen on the next page.

```
def sort_in_place(to_sort)
  # index starts at one, we can skip the first element, since we
  # would otherwise take it and place it in the first position,
  # which it already is
  for index in 1..(to_sort.length - 1)
```

```
    for inner_index in 0..(index - 1)
      if to_sort[inner_index] >= to_sort[index]
        to_sort.insert(inner_index, to_sort[index])
        to_sort.delete_at(index + 1)
      end
    end
  end

  return to_sort
end
```

Merge Sort | Runtime: O(n Log(n))

Merge sort divides an array in half, sorts each of those halves, then it merges them back together. Each halve has the same sorting algorithm applied to it. Eventually you are merging just two single-element arrays, with the "merge" part of the function doing the heavy lifting. The merge method operates by copying all the elements from the target array segment into a helper array, keeping track of where the start of the left and right halves should be. An example of this is visible below.

```
class MergeSort
  def sort(to_sort)
    # if the array is of length 0 or 1, consider it is already
    # sorted
    if to_sort.length <= 1
      then return to_sort
    end

    # otherwise split the remaining elements in two
    second_array = to_sort.slice!((to_sort.length / 2.0)
    .round..to_sort.length)

    # recursive method call on both arrays
    first_sorted_array = sort(to_sort)
    second_sorted_array = sort(second_array)

    # merge the two sorted arrays together
    return merge(first_sorted_array, second_sorted_array)
  end

private
  def merge(first_array, second_array)
    # if either array is empty consider the other already sorted
    return second_array if first_array.empty?
    return first_array if second_array.empty?
```

```
    # remove the smallest element out of the two arrays
    if first_array.first <= second_array.first
      element = first_array.shift
    else
      element = second_array.shift
    end

    # recursive call to construct the merged array
    return [element] + merge(first_array, second_array)
  end
end
```

Quicksort | Runtime: O(n Log(n)), O(n^2) worst case

Quicksort is one of the most popular types of sorting algorithms. It works by dividing the array to be sorted in two. An element called a "pivot" is chosen. This element will be the one that determines how the array is next subdivided. Different implementations of quicksort can use different methods for determining which element is the pivot. This method will influence the performance of the algorithm itself. Choosing the first element as the pivot will result in bad performance on nearly sorted or already sorted arrays. Choosing the last element can result in similarly poor performance.

A better choice for implementation would be to choose the middle element or a random element. The scenario that you're attempting to avoid with your choice of the pivot element is one where the sub-lists (created by dividing the array) have only one element. This algorithm works in the following way:

1. Divides the array in two.
2. Chooses a pivot (in this case the first element of our divided array).
3. Using two indices, iterates on the sub array. One index will go from left to right, and the other right to left.
4. As the two indices iterate, swaps the elements greater than the pivot to the right side of the sub array and the elements lesser than the pivot to the left side of the array.
5. Moves the pivot where the two indices have crossed paths.

An example of this algorithm can be found below.

```
class QuickSort
  def sort(to_sort, index_of_pivot = 0, right_index =
  to_sort.length - 1)
    old_right_index = right_index
    left_index = index_of_pivot
```

```
    # stop the recursion if nothing to sort
    if left_index >= right_index then
      return to_sort
    end

    # partition operation
    # move both indexes towards the center until they cross over
    # when left index finds an element greater than pivot and
    # right index finds an element smaller than pivot swap them
    while left_index < right_index
      while to_sort[left_index] <= to_sort[index_of_pivot] and
      left_index < to_sort.length - 1
        left_index = left_index + 1
      end

      right_index = right_index - 1 until to_sort[right_index] <=
      to_sort[index_of_pivot]

      # swap both elements
      if left_index < right_index
        to_sort[left_index], to_sort[right_index] =
        to_sort[right_index], to_sort[left_index]
      end
    end

    # swap pivot
    to_sort[index_of_pivot], to_sort[right_index] =
    to_sort[right_index], to_sort[index_of_pivot]

    # recursively sort the sub arrays
    sort(to_sort, index_of_pivot, right_index - 1)
    sort(to_sort, left_index, old_right_index)

    return to_sort
  end
end
```

Sorting and Searching Interview Questions

1. Write a method to sort an array of strings so that all the anagrams are next to each other.
2. Imagine you have a 20 GB file with one string per line. Explain how you would sort that file.
3. Given an M x N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

Testing

Even if you're not applying for a testing-related position, good programmers are also able to create tests for their programs. Testing problems usually fall under one of four categories:

- Test a real world object
- Test a piece of software
- Write code to test a function
- Explain how to troubleshoot an existing issue

When asking you questions related to testing, interviewers are often looking trying to assess more than just your ability to write multiple test cases. Interviewers are trying to measure the following:

- Big picture understanding: Do you really understand the purpose of the software? Can you prioritize test cases properly? Do you understand which components of a piece of software are the most important to test.
- Seeing the whole for its subcomponents: Do you understand how software works in a greater ecosystem? Say you're asked to test Google Spreadsheets. It's important to test the functionality of Spreadsheets itself, but it's just as important to make sure the software integrates correctly with Gmail, plug-ins, etc.
- Thought organization: Are you able to approach a problem in a structured manner, or do you just abstractly pull ideas out of your head? If asked to come up with the test cases for a camera, good candidates will break down the parts into categories like "Taking Photos", "Image Management" and "Configuration".
- Practicality: Are the testing plans that you've created realistic? If a user reports that their software is crashing every time they open one specific image, and you instruct them to reinstall the software, that's generally not a practical solution.

Testing Real World Objects

Don't be surprised if your interviewer asks you to test a tangible object. For example, you might be given the question "How would you test a paperclip?".

Step 1: Who will use it? Why?

You need to question your interviewer on who is using the product and for what purpose. The answer to this question might drastically impact your response. They could easily say "the clip is being used by teachers, to hold papers together" or "the clip is being used by school children to bend into the shape of animals". The answer to this question will change how you handle the remaining questions.

Step 2: What are the use cases?

By making a list of use cases you can clearly determine the purpose of this software so you can have an idea of what to test. In the example of a paperclip, the use case might be limited to a single case of "fastening paper together in a way that doesn't damage the paper". In other situations, their might be multiple use cases.

Step 3: What are the bounds of use?

The bounds of use are the constraints of the object in use. For instance, the paperclip may have an upper bound of holding 30 sheets of a paper with no permanent damage, and 30-50 sheets with minimal damage. These bounds can extend to other environmental factors as well. Does the paperclip need to work in extreme cold or extreme heat?

Step 4: What are the stress or failure conditions?

No product is immune to failure, and because of that you need to incorporate failure conditions into your testing. It's important to discuss when it's appropriate for a product to fail, and what failure should result in.

As an example, if you were testing a laundry machine, you might decide that the machine should be able to handle at least 30 shirts or 30 pairs of pants. Loading slightly more than this (30-45) may result in minor failure, such as inadequate cleaning. Significantly overloading the machine's capacity may result in extreme failure. However, this extreme failure case should result in the machine not being able to turn on, opposed to flooding or catching on fire.

Step 5: How would you perform the testing?

It might also be relevant to discuss the details of performing the testing. As an example, if you need to make sure a chair can withstand normal usage for five years, it would be unfeasible to assume that you can actually place the chair in a home and wait five years to determine success. Instead, you need to determine what "normal" usage is (how many times does a chair get sat on in a year). You would then want a machine to carry out these tests, in addition to some manual testing.

Testing a Piece of Software

When asked to test a piece of software, try to keep in mind that testing virtual objects is very similar to testing objects in the real world. Note that software testing has two core components:

- Manual vs. Automated Testing: Ideally, we'd like to be able to automate everything but that's not realistic. It's far easier to manually test certain things

because some features are too qualitative for a computer to effectively examine (e.g. if a photograph contains a cat). An additional improvement of manual testing is that humans are able to recognize issues that they haven't been explicitly instructed to look for.

- Black Box Testing vs. White Box Testing: This distinction refers to the degree of access we have into the software. In black box testing, we're just given the software as-is and need to test it. With white box testing, we have additional programmatic access to test individual functions.

Just like when you're testing real-world objects, it's a best practice to follow a logical sequence of steps.

Step 1: Determine whether or not you're doing Black Box Testing or White Box Testing

Though this question can often be delayed to a later step, it's a best practice to get it out of the way early.

Step 2: Who are the users? Why are they using it?

A piece of software typically has one or more target users, and the features are designed with this in mind. As an example, if you're asked to test software for parental controls on a web browser, your target users include both parents (who are implementing the blocking) and children (who are the recipients of blocking). You may also have additional user roles beyond these two.

Step 3: What are the use cases?

In the software blocking scenario, the use cases of the parents include installing the software, updating controls, removing controls, and of course their own personal internet usage. For the children, the use cases include accessing "allowable" content as well as content that is prohibited.

Step 4: What are the bounds of use?

Now that we have the vague use cases defined, we need to figure out what exactly this means. What does it mean for a website to be blocked? Should just the prohibited page be blocked, or the entire website? Is the application supposed to dynamically determine, or "learn" what prohibited content is, or is it on a black list of a white list? If it's supposed to learn what content is prohibited and what is acceptable, what degree of false positives or false negative is acceptable?

Step 5: What are the stress conditions and failure conditions?

When the software inevitably fails what should the failure look like? Clearly, the software failure shouldn't crash the computer. Instead, it's likely that the software should just permit a blocked site, or ban an allowable site. In the latter condition, you might want to discuss the possibility of a selective override with a password provided by the parent user.

Step 6: What are the test cases? How would you perform the testing?

The third and fourth steps should have roughly defined the use cases. In this step we further define them and discuss how to perform each one. What exact situations are you testing? Which of these steps can be automated? Which require manual intervention?

Testing a Function

In many ways, testing a function is the easiest type of testing. The conversation is typically briefer and less vague, as the testing is usually limited to validating input and output. Suppose you were asked to write code to test `sort(int, array)` which sorts an array of integers.

Step 1: Define the test cases

In general, you should think about the following categories of test cases:

- The normal case: Does it generate the correct output for typical inputs? Remember to think about the potential issues here. For example, because sorting often requires some sort of partitioning, it's reasonable to think that the algorithm might fail on arrays with an odd number of elements, since they can't be evenly partitioned. Your test case should list both examples.
- The extremes: What happens when you pass in an empty array? Or a very small, one element, array? What if you pass in a very large one?
- Nulls and "illegal" input: It is worthwhile to think about how the code should behave when given illegal input. For example, if you're testing a function to generate the nth Fibonacci number, your test cases should probably include the situation where n is negative.
- Strange input: This type of input occurs when a function is given an unusual input. For example, what if a function that is intended to sort arrays is given a pre-sorted array?

Creating these tests does require knowledge of the function you are writing. If you are unclear as to the constraints, you will need to ask your interviewer about this first.

Step 2: Define the expected result

Often, the expected result is obvious: the correct output. However, in some cases, you might want to validate additional aspects. For instance, if the sort method returns a new, sorted copy of the array, it would be prudent for you to ensure that the original array has remained untouched.

Step 3: Write the test code

Once you have the test cases and results defined, writing the code to implement the test cases should be fairly straightforward.

Troubleshooting Questions

A final type of question is explaining how you would debug or troubleshoot an existing issue. You should approach this type of question in a structured manner, like anything else. Here's an example: You're working on the Google team responsible for Chrome and you receive a bug report "Chrome crashes on launch". What do you do? You could reinstall the software, but that wouldn't fix the problem for all of the other users.

Step 1: Understand the Scenario

Your first step should be to ask questions to understand as much about the situation as possible. Some example questions might include:

- How long has the user been experiencing this issue?
- What version of the browser is it? What operating system?
- Does the issue happen consistently, or how often does it happen? When does it happen?
- Is there an error report that launches?

Step 2: Break Down the Problem

Once you understand the details of the scenario, you want to break down the problem into testable units. The flow of the situation might be something like the following:

1. Go to the Windows Start menu.
2. Click on the Chrome icon.
3. Browser instance starts.
4. Browser loads settings.
5. Browser issues HTTP request for homepage.
6. Browser gets HTTP response.
7. Browser parses webpage.
8. Browser displays content.

Step 3: Create Specific, Manageable Tests

Each of the above components should have realistic instructions, or things that you could ask a user to do or things you can try yourself. In a real-world scenario you will be dealing with customers and you can't give them instructions that they can't or won't do.

Testing Interview Questions

1. Find the mistake(s) in the following Ruby code:
```
function(first_name, last_name) {
  Name = first_name + last_name;
  return Name;
}
```
2. You are given the source to an application that crashes when it is run. After running it ten times in a debugger, you find it never crashes in the same place. The application is written in Ruby. What programming errors could be causing this crash? How would you test each one?
3. How would you load a test webpage without using any test tools?
4. How would you test a pen?
5. How would you test an ATM at a bank?

Knowledge Based Interview Questions

One of the best ways to increase your preparedness for technical interview questions is by practicing answering them by using a question bank. The above Smarterer tests are also a good source of questions to prepare with. These questions are in no particular order and vary significantly in their difficulty. Interviewers will not expect you to know the answers to all of these questions, as some of them focus on advanced concepts that are typically reserved for intermediate-level developers, but knowing how to answer them will impress your interviewers. These questions are denoted with an asterisk.

Ruby and Ruby on Rails

1. What are Ruby gems?
2. What's the difference between a Symbol and a String?
3. What is the purpose of yield?
4. What are class variables? How do you define them?
5. How do you define instance variables?
6. How do you define global variables?
7. Does Ruby support constructors? How are they declared?
8. How can you dynamically define a method body?*
9. What is a Range?
10. How can you implement method overloading?*
11. What is the difference between '&&', 'and' and '&' operators?
12. How can you create setter and getter methods in Ruby?
13. What is the convention for using "!" at the end of a method name?
14. What is a module?
15. Does Ruby support multiple inheritance?
16. How can you achieve the same effect as multiple inheritance in Ruby? What is a mixin?
17. How can you implement a singleton pattern?
18. How can you implement an observer pattern?*
19. What is the purpose of the environment.rb and application.rb files?
20. How can you define a constant?
21. How can you trigger a method when a module is included inside a class?
22. What is the default access modifier (public/protected/private) for a method?
23. How can you call the base class method from inside of its overridden method?
24. Define the Rails MVC implementation using an example.
25. What is a migration?
26. What are some of the methods available to migrations?
27. What can I find in the schema.rb file?
28. Tell me some of the types of validations and their intended use.
29. Tell me some of the types of callbacks and their intended use.
30. What are all of the types of associations? When would I use each one?

31. What's the purpose of the id column in a database?
32. Why would I use a polymorphic association?
33. What would the following queries return:
    a. `client = Client.find(10)`
    b. `Client.order("created_at DESC")`
    c. `Client.limit(5).offset(30)`
34. What are the HTTP verbs, paths, action and usage created by including the following in your routes.rb file? `resources :photos`
35. What is the params hash and how do you retrieve data from it?
36. How can you accept parameters in JSON?
37. What is a session and how does it work?
38. How does a before_filter work?
39. How is calling render different from calling redirect_to?
40. How do you render JSON?
41. What are asset tag helpers and what do they do?
42. What does yield do and how do I use it?
43. What is a partial used for? How do I name the files?
44. How is a form helper used?
45. What are some inputs for forms and what is the Rails method for creating them?
46. What are some of the ways that Active Support extends the functionality of Ruby?
47. How does the Rails Asset Pipeline work? When was it introduced?
48. What is the correct way to reference a file in the asset pipeline?
49. What is a preprocessor and how does it work?
50. What are some of the ways that you can configure your Rails generators? Why would someone do this?
51. What are some new features in Rails 4?
52. What is becoming deprecated in Rails 4?
53. Why are you excited about Rails 4?
54. What is scope?
55. Can you give an example of a class that should be inside the lib folder?
56. Where should you put code that is supposed to run when your application launches?
57. Can you give me an example of a tool that's used specifically for deployment?
58. How can you migrate your database schema one level down?
59. What is a sweeper?*
60. How can you implement caching in Rails?
61. What is a filter? When is it called?
62. What do controllers do in Rails?
63. What is RESTful routing?
64. How can you list all routes for an application?
65. How can you send a multi-part email?
66. What is the purpose of layouts?

67. Is it possible to embed partial views inside layouts? How?
68. What is Rake?
69. What is Capistrano?
70. What is a “has and belongs to many” (HABTM) association?
71. What is the difference between has_one and belongs_to?
72. How can you implement single table inheritance?
73. What is a polymorphic association?
74. What is eager loading?
75. How can you eager load associated objects?*
76. How does validation work?
77. How can you add a custom validation on your model?
78. What is “Flash”?
79. How can you install the missing gems that your application requires in the simplest way?
80. How can you implement internationalization?
81. What plugin would you recommend for authentication and authorization?
82. What plugin do you use for full-text search?
83. What is the difference between a plugin and a gem?
84. How can you implement a search feature that searches for multiple models?*
85. How can you upload a file to a server?
86. Is Rails scalable?*
87. How can you secure a Rails application?
88. What are some good community resources for Rails?
89. Where would you ask the community to get answers for your technical questions?
90. Explain the difference between Controller & Model logic in Rails

JavaScript

1. What are the three layers of progressive enhancement?
2. Why do you generally place JavaScript (JS) at the bottom of an HTML document?
3. What is the most widely supported method of persistent data storage in JS?
4. What is an anonymous function?
5. When is a function considered to be a callback function?
6. How do you obtain feedback from a user?
7. What are four types of DOM nodes?
8. Explain what the DOM is and how it relates to HTML and JS?
9. What happens if you try to set a node attribute for an attribute that does not exist on the node already?
10. What are the three data types you can use with a variable?
11. What is an associative array?
12. What is a multidimensional array?
13. Why is it best to position all variables at the top of your JS file?
14. Why are some of the words reserved in JS?
15. How are anonymous functions different from basic functions?
16. What is the difference between an event handler and an event listener?
17. What method is used for fallback support of event listeners in IE8 and earlier?
18. What is the purpose of the `preventDefault()` method?
19. Why shouldn't you start a function or variable name with an underscore?
20. When is it best to use an anonymous function?
21. What is the data type of 3.14?
22. What is the data type of 3?
23. The function `Math.sqrt(-5);` returns `NaN`. What does `NaN` mean?
24. Write a function that calculates `Math.sqrt(-5);` and evaluates to `NaN` and false otherwise.
25. What does the following statement evaluate to? `NaN === NaN`
26. Write a function to round 3.78 down to the nearest integer
27. Write a function to calculate the length of the string "bob".
28. Are JavaScript strings mutable?
29. Write a function to convert all the letters of the string "Happy New Year" to uppercase.
30. What are falsy values? What are falsy values in JavaScript?
31. Write a ternary operator that prints "Oh Yea" to the console if 5 is greater than 3 and prints "Huh" otherwise.
32. Show to ways to access the height of the following "Shaq" object.

```
var shaq = {
  body: {
    height: "7'1",
    weight: 325
```

```
  },
  stats: {
    points_per_game: 24,
    minutes_per_game: 11
  }
}
```

jQuery

1. Create an HTML page. Using JavaScript, set variable `a` to 3 and variable `b` to 5 and display the sum of `a` and `b` in the browser.
2. Create a box that gives the user the message "Hello World!" and makes the user click "OK" before the rest of the page loads.
3. Write the text "Hello World" in JavaScript in a way that preserves the CSS formatting related to the <p> tag.
4. Fade the <p> tag below on to the screen over the course of three seconds

```
<!DOCTYPE html>
<html>
  <head>
    <title>Fade In</title>
  </head>
  <body>
    <p>This text will fade in over three seconds</p>
  </body>
</html>
```

5. Create a page that asks the user "What is your name?" and prints "Welcome NAME" where NAME is the name supplied by the user.
6. Create an array of four economists: "Taylor", "Mankiw", "Roubini", "Krugman". Iterate over the array and print the names in the browser.
7. Write a function that prints out the current date in mm/dd/yyyy format.
8. Create a function print, so that `print("Hello World")` will render the text "Hello World" in the browser.
9. Add the tag `<p> JavaScript drives me crazy sometimes</p>` to the following div:

```
<div class="intro">
  <h1>Mental Status?</h1>
</div>
```

10. Add the text "Sup" to the beginning of `<h1>My homies</h1>`.
11. Delete `<h3>Delete me</h3>`.
12. Replace `<h4>This is the old h4</h4>` with `<h4>This is the BRAND NEW h4 </h4>`.
13. Remove the "title" class from `<p class="title">Breaking Bad</p>`

14. Add a solid red border around `<p id=”block”>jQuery will make this have a solid red border</p>`.
15. Make the following formatting changes to the div below: background color FF7373, font color white, border black

```
<div id = "meow">
  <p>CSS is FUN</p>
</div>
```

16. Store the value of the src attribute of `<img id = “chapel” src=”indian_chapel.jpg” height = ‘300px’ width = ‘400px’ />` in a variable to the screen to demonstrate it has been done correctly.
17. Change the picture source of `<img id = “picture” src=”indian_chapel.jpg” height = ‘300px’ width = ‘400px’ />` to be “oktoberfest.jpg”
18. Make the following image gradually disappear over a period of 3 seconds.
```
<img id = "picture" src="oktoberfest.jpg" height = '300px'
width = '400px'/>
```

19. Create a page with two images. For each image in the page, give the user the alert “I found an image”.
```
<img src="oktoberfest.jpg" height = "300px" width =
"400px"/>
  <br />
<img src="indian_chapel.jpg" height = "300px" width =
"400px"/>
```

20. Make a paragraph that says "This will be changed" move 650 pixels to the right, grow to 50px font size, and change opacity to 0.5 over the course of 1.5 seconds.
```
<body>
  <div id ="message">
    <p>This will be changed</p>
  </div>
</body>
```

Git

1. How do you start Git and what is the process called?
2. How do you determine the current state of the project?
3. How do you move all your changes since your last commit to the staging area?
4. Store the saved changes in the repository and add a message "first commit".
5. Show a list of all the commits that have been made.
6. How do you put your local repository (repo) on the GitHub server?
7. Take a git project from GitHub to your local machine to work on it.
8. What is the purpose of working off multiple branches.
9. Create a branch called working.
10. List all of the branches.
11. Switch to the working branch.
12. Merge the changes that were made in the working branch with the master branch.
13. Delete the working branch.
14. Put everything up on the remote repository.
15. Pull a branch from a remote repository and create the branch locally.