

The Five Essential Phone-Screen Questions

Stevey's Drunken Blog Rants™

I've been on a lot of SDE interview loops lately where the candidate failed miserably: not-inclined votes all around, even from the phone screeners who brought the person in initially.

It's usually pretty obvious when the candidate should have been eliminated during the phone screens. Well, it's obvious in retrospect, anyway: during the interviews, we find some horrible flaw in the candidate which, had anyone thought to ask about it during the phone screen, would surely have disqualified the person.

But we didn't ask. So the candidate came in for interviews and wound up wasting everyone's time.

Antipatterns

I've done informal postmortems on at least a hundred phone screens, many of them my own. Whenever a candidate bombs the interviews, I want to know what went wrong with the screen. And guess what? A pattern has emerged. Two patterns, actually.

The first pattern is that for most failed phone screens, *the candidate did most of the talking*. The screener only asked about stuff on the candidate's resume, and the candidate was able to talk with passion and enthusiasm about this incredibly cool thing they did, blah blah blah, and the screener was duly impressed.

That's how many/most phone screens go wrong.

The right way to do a phone screen is to do most of the talking, or at least the driving. You look for specific answers, and you guide the conversation along until you've got the answer or you've decided the candidate doesn't know it. Whenever I forget this, and get lazy and let the candidate drone on about their XML weasel-pin connector project, I wind up bringing in a dud.

The second pattern is that *one-trick ponies only know one trick*. Candidates who have programmed mostly in a single language (e.g. C/C++), platform (e.g. AIX) or framework (e.g. J2EE) usually have major, gaping holes in their skills lineup. These candidates will fail their interviews here because our interviews cover a broad range of skill areas.

These two phone screen (anti-)patterns are related: if you only ask the candidate about what they know, you've got a fairly narrow view of their abilities. And you're setting yourself up for a postmortem on your phone screen.

Acid Tests

In an effort to make life simpler for phone screeners, I've put together this list of Five Essential Questions that you need to ask during an SDE screen. They won't guarantee that your candidate will be great, but they will help eliminate a huge number of candidates who are slipping through our process today.

These five areas are litmus tests -- very good ones. I've chosen them based on the following criteria:

- 1) *They're universal* - every programmer needs to know them, regardless of experience, so you can use them in *all* SDE phone screens, from college hires through 30-year veterans.
- 2) *They're quick* - they're areas that you can probe very quickly, without eating too much into your phone-screen time. Each area can be assessed with 1 to 5 minutes of "weeder questions", and each area has almost unlimited weeder questions to choose from.
- 3) *They're predictors* - there are certain common "SDE profiles" that are easy to spot because they tend to fail (and I mean *really* fail) in one or more of these five areas. So the areas are amazingly good at weeding out bad candidates.

You have to probe all five areas; you can't skip any of them. Each area is a proxy for a huge body of knowledge, and failing it very likely means failing the interviews, even though the candidate did fine

in the other areas.

Without further ado, here they are: The Five Essential Questions for the first phone-screen with an SDE candidate:

- 1) **Coding.** The candidate has to write some simple code, with correct syntax, in C, C++, or Java.
- 2) **OO design.** The candidate has to define basic OO concepts, and come up with classes to model a simple problem.
- 3) **Scripting and regexes.** The candidate has to describe how to find the phone numbers in 50,000 HTML pages.
- 4) **Data structures.** The candidate has to demonstrate basic knowledge of the most common data structures.
- 5) **Bits and bytes.** The candidate has to answer simple questions about bits, bytes, and binary numbers.

Please understand: what I'm looking for here is a *total vacuum* in one of these areas. It's OK if they struggle a little and then figure it out. It's OK if they need some minor hints or prompting. I don't mind if they're rusty or slow. What you're looking for is candidates who are utterly clueless, or horribly confused, about the area in question.

For example, you may find a candidate who decides that a Vehicle class should be a subclass of ParkingGarage, since garages contain cars. This is just busted, and it's un-fixable in any reasonable amount of training time.

Or a candidate might decide, when asked to search for phone numbers in a bunch of text files, to write a 2000-line C++ program, at which point you discover they've never heard of "grep", or at least never used it.

When a candidate is totally incompetent in one of these Big Five areas, the chances are very high that they'll bomb horribly when presented with our typical interview questions. Last week I interviewed an SDE-2 candidate who made both of the mistakes above (a vehicle inheriting from garage, and the 2000-line C++ grep implementation.) He was by no means unusual, even for the past month. We've been bringing in *many* totally unqualified candidates.

The rest of this document describes each area in more detail, and gives example questions, and solutions.

Area Number One: Coding

The candidate has to write some code. Give them a coding problem that requires writing a short, straightforward function. They can write it in whatever language they like, as long as they don't just call a library function that does it for them.

It should be a trivial problem, one that even a slow candidate can answer in 5 minutes or less.

(If the candidate seems insulted by the thought of having to get their hands dirty with a trivial coding question, after all their years of experience, patents, etc., tell them it's required procedure and ask them to humor you. If they refuse, tell them we only interview people who can demonstrate coding skills over the phone, thank them for their time, and end the call.)

Give them a few minutes to write and hand-simulate the code. Tell them they need to make it syntactically correct and complete. Make them read the code to you over the phone. Copy down what they read back. Put it into your writeup. If they're sloppy, or don't want to give you exact details, give them one more chance to correct it, and then go with Not Inclined.

(Note added 10/6/04) -- another good approach being used by many teams is to give the candidate "homework". E.g. you can give them an hour to solve some coding problem (harder than the ones below) and email the solution to you. Works like a charm. Definitely preferable to reading code over the phone.

Anyway, here are some examples. I've given solutions in Java, mostly. I've gone back and forth on accepting solutions in other languages (e.g. Ruby, Perl, Python), and I've decided that candidates need to be able to code their answers in C, C++ or Java. It's wonderful if they know other languages, and in fact those who do tend to do a *lot* better overall. But to be an Amazon SDE, you need to prove you can do C++ or Java first.

Example 1: *Write a function to reverse a string.*

Example **Java** code:

```
public static String reverse ( String s ) {
    int length = s.length(), last = length - 1;
    char[] chars = s.toCharArray();
    for ( int i = 0; i < length/2; i++ ) {
        char c = chars[i];
        chars[i] = chars[last - i];
        chars[last - i] = c;
    }
    return new String(chars);
}
```

Example output for "Madam, I'm Adam": madA m'I ,madaM

Example 2: Write function to compute Nth fibonacci number:

Java and **C/C++**:

```
static long fib(int n) {
    return n <= 1 ? n : fib(n-1) + fib(n-2);
}
```

(Java Test harness)

```
public static void main ( String[] args ) {
    for ( int i = 0; i < 10; i++ ) {
        System.out.print ( fib(i) + ", " );
    }
    System.out.println ( fib(10) );
}
```

(C/C++ Test Harness)

```
main () {
    for ( int i = 0; i < 10; i++ ) {
        printf ( "%d, ", fib(i) );
    }
    printf ( "%d\n", fib(10) );
}
```

Test harness output:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

Example 3: Print out the grade-school multiplication table up to 12x12

Java: (similar for **C/C++**)

```
public static void multTables ( int max )
{
    for ( int i = 1; i <= max; i++ ) {
        for ( int j = 1; j <= max; j++ ) {
            System.out.print ( String.format ( "%4d", j * i ));
        }
        System.out.println();
    }
}
```

Example output:

1	2	3	4	5	6	7	8	9	10	11	12
2	4	6	8	10	12	14	16	18	20	22	24
3	6	9	12	15	18	21	24	27	30	33	36
4	8	12	16	20	24	28	32	36	40	44	48
5	10	15	20	25	30	35	40	45	50	55	60
6	12	18	24	30	36	42	48	54	60	66	72
7	14	21	28	35	42	49	56	63	70	77	84
8	16	24	32	40	48	56	64	72	80	88	96
9	18	27	36	45	54	63	72	81	90	99	108

10	20	30	40	50	60	70	80	90	100	110	120
11	22	33	44	55	66	77	88	99	110	121	132
12	24	36	48	60	72	84	96	108	120	132	144

Example 4: Write a function that sums up integers from a text file, one int per line.

Java:

```
public static void sumFile ( String name ) {
    try {
        int total = 0;
        BufferedReader in = new BufferedReader ( new FileReader ( name ) );
        for ( String s = in.readLine(); s != null; s = in.readLine() ) {
            total += Integer.parseInt ( s );
        }
        System.out.println ( total );
        in.close();
    }
    catch ( Exception xc ) {
        xc.printStackTrace();
    }
}
```

Example 5: Write function to print the odd numbers from 1 to 99.

C/C++:

```
void printOdds() {
    for (int i = 1; i < 100; i += 2) {
        printf ("%d\n", i); // or cout << i << endl;
    }
}
```

Java:

```
public static void printOdds() {
    for (int i = 1; i < 100; i += 2) {
        System.out.println ( i );
    }
}
```

Example 6: Find the largest int value in an int array.

Java:

```
public static int largest ( int[] input ) {
    int max = Integer.MIN_VALUE;
    for ( int i = 0; i < input.length; i++ ) {
        if ( input[i] > max ) max = input[i];
    }
    return max;
}
```

Example 7: Format an RGB value (three 1-byte numbers) as a 6-digit hexadecimal string.

Java:

```
public String formatRGB ( int r, int g, int b ) {
    return (toHex(r) + toHex(g) + toHex(b)).toUpperCase();
}

public String toHex ( int c ) {
    String s = Integer.toHexString ( c );
    return ( s.length() == 1 ) ? "0" + s : s;
}
```

Or in Java 1.5:

```
public String formatRGB ( int r, int g, int b ) {
    return String.format ( "%02X%02X%02X", r, g, b );
}
```

}

Example output for (255, 0, 128):

You can ask any question you like; doesn't have to be one of the ones above. They're just examples.

Some properties of a good weeder phone-screen coding question are:

1. *It's simple.* It has to be something that you should be able to solve, trivially, in about 2 minutes or less. Not too tricky. Basic stuff.
2. *You've solved it.* You shouldn't ask a question unless you've solved it yourself recently, so you know it's a reasonable question, and you can evaluate their answer to it. You should consider coding it yourself during the time you've given them to do it.
3. *It has loops or recursion.* Recursion is actually preferable. Being able to reason recursively or inductively is important for many areas of computing, including using hierarchical data representations (e.g. XML), distributed computing, searching, and sorting. Many candidates simply can't think recursively, and this often goes undetected until interview-time. Try to find out at compile-time! Er, phone-screen time, that is.
4. *It has formatted output.* This is a basic skill, useful for debugging, simple report generation, and lots of other things. "printf" is a universal standard; it exists in C, C++, Java, Perl, Ruby, Python, and virtually every other mainstream language, at *least* as a library call. Like file I/O, it's a good indicator as to whether the candidate has written "real" code before.
5. *It has text-file I/O.* Candidates who have worked in frameworks for too long often become unable to function as programmers outside that framework. Not being able to do simple file I/O is a common indicator that they've grown overly dependent on a particular framework.

It's hard to cover *all* these things and still be a short weeder question. If you think of a question that has all these properties, let me know.

Area Number Two: Object-Oriented Programming

We shouldn't hire SDEs (arguably excepting college hires) who aren't at least somewhat proficient with OOP. I'm not claiming that OOP is good or bad; I'm just saying you have to know it, just like you have to know the things you can and can't do at an airport security checkpoint.

Two reasons:

- 1) OO has been popular/mainstream for more than 20 years. Virtually every programming language supports OOP in some way. You can't work on a big code base without running into it.
- 2) OO concepts are an important building block for creating good service interfaces. They represent a shared understanding and a common vocabulary that are sometimes useful when talking about architecture.

So you have to ask candidates some OO stuff on the phone.

a) Terminology

The candidate should be able to give satisfactory definitions for a random selection of the following terms:

1. class, object (and the difference between the two)
2. instantiation
3. method (as opposed to, say, a C function)
4. virtual method, pure virtual method
5. class/static method
6. static/class initializer
7. constructor
8. destructor/finalizer
9. superclass or base class
10. subclass or derived class
11. inheritance
12. encapsulation
13. multiple inheritance (and give an example)

14. delegation/forwarding
15. composition/aggregation
16. abstract class
17. interface/protocol (and different from abstract class)
18. method overriding
19. method overloading (and difference from overriding)
20. polymorphism (without resorting to examples)
21. is-a versus has-a relationships (with examples)
22. method signatures (what's included in one)
23. method visibility (e.g. public/private/other)

These are just the bare basics of OO. Candidates should know this stuff cold. It's not even a complete list; it's just off the top of my head.

Again, I'm not advocating OOP, or saying anything about it, other than that it's ubiquitous so you have to know it. You can learn this stuff by reading a single book and writing a little code, so no SDE candidate (except maybe a brand-new college hire) can be excused for not knowing this stuff.

I draw a distinction between "*knows it*" and "*is smart enough to learn it*." Normally I allow people through for interviews if they've got a gap in their knowledge, as long as I think they're smart enough to make it up on the job.

But for these five areas, I expect candidates to know them. It's not just a matter of being smart enough to learn them. There's a certain amount of common sense involved; I can't imagine coming to interview at Amazon and not having brushed up on OOP, for example. But these areas are also so fundamental that they serve as real indicators of how the person will do on the job here.

b) OO Design

This is where most candidates fail with OO. They can recite the textbook definitions, and then go on to produce certifiably insane class designs for simple problems. For instance:

- They may have Person multiple-inherit from Head, Body, Arm, and Leg.
- They may have Car and Motorcycle inherit from Garage.
- They may produce an elaborate class tree for Animals, and then declare an enum ("Lion = 1, Bear = 2", etc.) to represent the type of each animal.
- They may have exactly one static instance of every class in their system.

(All these examples are from real candidates I've interviewed in the past 3 weeks.)

Candidates who've only studied the terminology without ever doing any OOP often don't really get it. When they go to produce classes or code, they don't understand the difference between a static member and an instance member, and they'll use them interchangeably.

Or they won't understand when to use a subclass versus an attribute or property, and they'll assert firmly that a car with a bumper sticker is a subclass of car. (Yep, 2 candidates have told me that in the last 2 weeks.)

Some don't understand that objects are supposed to know how to take care of themselves. They'll create a bunch of classes with nothing but data, getters, and setters (i.e., basically C structs), and some Manager classes that contain all the logic (i.e., basically C functions), and voila, they've implemented procedural programming perfectly using classes.

Or they won't understand the difference between a char*, an object, and an enum. Or they'll think polymorphism is the same as inheritance. Or they'll have any number of other fuzzy, weird conceptual errors, and their designs will be fuzzy and weird as well.

For the OO-design weeder question, have them describe:

1. What classes they would define.
2. What methods go in each class (*including signatures*).
3. What the class constructors are responsible for.
4. What data structures the class will have to maintain.
5. Whether any Design Patterns are applicable to this problem.

Here are some examples:

1. Design a deck of cards that can be used for different card game applications.

Likely classes: a Deck, a Card, a Hand, a Board, and possibly Rank and Suit. Drill down on who's responsible for creating new Decks, where they get shuffled, how you deal cards, etc. Do you need a different instance for every card in a casino in Vegas?

2. Model the Animal kingdom as a class system, for use in a Virtual Zoo program.

Possible sub-issues: do they know the animal kingdom at all? (I.e. common sense.) What properties and methods do they immediately think are the most important? Do they use abstract classes and/or interfaces to represent shared stuff? How do they handle the multiple-inheritance problem posed by, say, a tomato (fruit or veggie?), a sponge (animal or plant?), or a mule (donkey or horse?)

3. Create a class design to represent a filesystem.

Do they even know what a filesystem is, and what services it provides? Likely classes: Filesystem, Directory, File, Permission. What's their relationship? How do you differentiate between text and binary files, or do you need to? What about executable files? How do they model a Directory containing many files? Do they use a data structure for it? Which one, and what performance tradeoffs does it have?

4. Design an OO representation to model HTML.

How do they represent tags and content? What about containment relationships? Bonus points if they know that this has already been done a bunch of times, e.g. with DOM. But they still have to describe it.

The following commonly-asked OO design interview questions are probably too involved to be good phone-screen weeders:

1. Design a parking garage.
2. Design a bank of elevators in a skyscraper.
3. Model the monorail system at Disney World.
4. Design a restaurant-reservation system.
5. Design a hotel room-reservation system.

A good OO design question can test coding, design, domain knowledge, OO principles, and so on. A good weeder question should probably just target whether they know when to use subtypes, attributes, and containment.

Area Number Three: Scripting and Regular Expressions

Many C/C++/Java candidates, even some with 10+ years of experience, would happily spend a week writing a 2,500-line program to do something you could do in 30 seconds with a simple Unix command.

I now pose the following question to ALL candidates, whether on the phone or in an interview, because it eliminates so many of them:

Last year my team had to remove all the phone numbers from 50,000 Amazon web page templates, since many of the numbers were no longer in service, and we also wanted to route all customer contacts through a single page.

Let's say you're on my team, and we have to identify the pages having probable U.S. phone numbers in them. To simplify the problem slightly, assume we have 50,000 HTML files in a Unix directory tree, under a directory called "/website". We have 2 days to get a list of file paths to the editorial staff. You need to give me a list of the .html files in this directory tree that appear to contain phone numbers in the following two formats: (xxx) xxx-xxxx and xxx-xxx-xxxx.

How would you solve this problem? Keep in mind our team is on a short (2-day) timeline.

Here are some facts for you to ponder:

1. Our Contact Reduction team *really did* have exactly this problem in 2003. This isn't a made-up example.
2. Someone on our team produced the list within an hour, and the list supported more than just the 2 formats above.

- About 25% to 35% of all software development engineer candidates, independent of experience level, *cannot solve this problem*, even given the entire interview hour and lots of hints.

I take as much time as necessary to explain the problem to candidates, to ensure that they understand it and can paraphrase the problem requirements correctly.

For the record, I'm not being tricky here. Once candidates start down the wrong path (i.e. writing a gigantic C++ program to open every file and parse character by character, using a home-grown state machine), I stop them, tell them this will take too long, and ask if there are any other possibilities. I ask if there are any tools or utilities that might be of use. I give them plenty of hints, and ultimately I tell them the answer.

Even after I tell them the answer, they often *still* don't get it.

Here's one of many possible solutions to the problem:

```
grep -l -R --perl-regexp "\b(\{3\}\s*|\{3\}-\{3\}-\{4\}\b" * > output.txt
```

But I don't even expect candidates to get that far, really. If they say, after hearing the question, "Um... grep?" then they're probably OK. I can ask them for the approximate syntax for the regular expression to use, and as long as they have a reasonable clue, I'm fine with it. Heck, if they can tell me where they'd *look* to find the syntax, I'm fine with it.

They can also use `find`, or write a Perl script (or awk or bash or etc.). Anything that shows they have even the tiniest inkling of why Unix is *Unix*.

They can even write a Java or C++ program, provided they can actually write an entire working program in, say, half an hour or less, on the board, or at least convince me that they will get it working quickly. But I've only ever had that happen once; an insanely good C++ programmer burned through a 175-line C++ program on the whiteboard that more or less solved it. We made him an offer. But usually they throw in the towel when they find out they have to remember how to do file I/O, or traverse a directory tree.

For what it's worth, this failure mode is unique to Java and C/C++ programmers. Perl programmers laugh and solve it in 30 seconds or less. I have some easy questions that make Perl programmers cry, but this isn't one of them.

In my experience, a programmer who only knows one language (where C and C++ count as one language for this exercise) is usually completely lost in one of these Five Essential Areas.

You don't necessarily have to ask the HTML phone-number question. Another one I used to ask, one that worked equally well, was:

Let's say you're on my team, and I've decided I'm a real stickler for code formatting. But I've got peculiar tastes, and one day I decide I want to have all parentheses stand out very clearly in your code.

So let's say you've got a set of source files in C, C++, or Java. Your choice. And I want you to modify them so that in each source file, every open- and close-paren has exactly one space character before and after it. If there is any other whitespace around the paren, it's collapsed into a single space character.

For instance, this code:

```
foo (bar ( new Point(x, graph.getY()) ));
```

Would be modified to look like this:

```
foo ( bar ( new Point ( x, graph.getY ( ) ) ) ) ;
```

I tell you (as your manager) that I don't care how you solve this problem. You can take the code down to Kinko's Copies and manually cut and paste the characters with scissors if you like.

How will you solve this problem?

Same thing, more or less. You'd do it with a Unix command like `sed` (using a regular expression), or do it in your editor using a regex, or write a quick Ruby script, whatever. I'd even accept having

them use a source-code formatter, *provided* they can tell me in detail how to use it, during the interview (to a level of detail that convinces me they've used it before.)

There are all sorts of variations on this problem. Generally you want to come up with a real-life scenario that involves searching text files for patterns, and see if the candidate wants to solve it by writing a giant chunk of C++ or Java code.

Area Number Four: Data Structures

SDE candidates need to demonstrate a basic understanding of the most common data structures, and of the fundamentals of "big-O" algorithmic complexity analysis.

Here's what they need to know about big-O. They need to know that algorithms usually fall into the following performance classes: constant-time, logarithmic, linear, polynomial, exponential, and factorial.

For the standard data structures in java.util, STL, or those built into a higher-level language, they need to know the big-O complexity for the operations on those data structures. *Example:* they should know that finding an element in a hashtable is usually constant-time, that finding an element in a balanced binary tree is order $\log(n)$, that finding an element in a linked list is order N , and that finding an element in a sorted array is order $\log(n)$. Similarly for insert/update/delete operations.

And they should be able to explain why each operation falls into a particular complexity class. For instance: "Computing a hash value doesn't depend on the number of items in the hashtable." Or: "you have to search the entire linked list, even if it's sorted, to find an arbitrary element in it." No math needed, no proofs, just explanations.

The (concrete) data structures they absolutely must understand are these:

- 1) *arrays* - I'm talking about C-language and Java-language arrays: fixed-sized, indexed, contiguous structures whose elements are all of the same type, and whose elements can be accessed in constant time given their indices.
- 2) *vectors* - also known as "growable arrays" or ArrayLists. Need to know that they're objects that are backed by a fixed-size array, and that they resize themselves as necessary.
- 3) *linked lists* - lists made of nodes that contain a data item and a pointer/reference to the next (and possibly previous) node.
- 4) *hashtables* - amortized constant-time access data structures that map keys to values, and are backed by a real array in memory, with some form of collision handling for values that hash to the same location.
- 5) *trees* - data structures that consist of nodes with optional data elements and one or more child pointers/references, and possibly parent pointers, representing a heirarchical or ordered set of data elements.
- 6) *graphs* - data structures that represent arbitrary relationships between members of any data set, represented as networks of nodes and edges.

There are, to be sure, many other important data structures one should know about, but not knowing about the six listed above is inexcusable, and grounds for rejection in a phone screen.

Candidates should be able to describe, for any of the data structures above:

- what you use them for (real-life examples)
- why you prefer them for those examples
- the operations they typically provide (e.g. insert, delete, find)
- the big-O performance of those operations (e.g. logarithmic, exponential)
- how you traverse them to visit all their elements, and what order they're visited in
- at least one typical implementation for the data structure

Candidates should know the difference between an abstract data type such as a Stack, Map, List or Set, and a concrete data structure such as a singly-linked list or a hash table. For a given abstract data type (e.g. a Queue), they should be able to suggest at least two possible concrete implementations, and explain the performance trade-offs between the two implementations.

Example weeder questions:

- 1) *What are some really common data structures, e.g. in java.util?*
- 2) *When would you use a linked list vs. a vector?*
- 3) *Can you implement a Map with a tree? What about with a list?*
- 4) *How do you print out the nodes of a tree in level-order (i.e. first level, then 2nd level, then 3rd level, etc.)*
- 5) *What's the worst-case insertion performance of a hashtable? Of a binary tree?*
- 6) *What are some options for implementing a priority queue?*

And so on. Just a few quick questions should cover this area, provided you don't focus exclusively on linear ordered sequences (lists, arrays, vectors and the like).

Area Number Five: Bits and Bytes

This area is fairly contentious, at least inasmuch as people who don't know this area claim you don't need to know it.

(*Hint*: that's true for everything. Nobody likes to admit they don't know something you need to know. I'll start: I should know more about math; it's inexcusable. I'm doing all kinds of stuff the long, slow, dumb way because of my rusty math skills. But at least I admit it, and I've been studying my math books semi-regularly in an attempt to repair my skills.)

Candidates *do* need to know about bits and bytes, at least at the level that I'm outlining here. Otherwise they're prone to having an integer-overflow error in their code that brings the website down and costs us millions. Or spending a week trying to decode a serialized object they're debugging. Or whatever. Computers don't have ten fingers; they have one. So people need to know this stuff.

Candidates should know what bits and bytes are. They should be able to count in binary; e.g. they should be able to tell you what 2^5 or 2^{10} is, in decimal. They shouldn't stare blankly at you when you ask with 2^{16} is. It's a special number. They should know it.

They should know at least the logical operations AND, OR, NOT, and XOR, and how to express them in their favorite/strongest programming language.

They should understand the difference between a bitwise-AND and a logical-AND; similarly for the other operations.

Candidates should know the probable sizes of the primitive data types for a standard 32-bit (e.g. Intel) architecture.

If they're a Java programmer, they should know exactly what the primitive types are (byte, short, int, long, float, double, char, boolean) and, except for boolean, exactly how much space is allocated for them per the Java Language specification.

Everyone should know the difference between signed and unsigned types, what it does to the range of representable values for that type, and whether their language supports signed vs. unsigned types.

Candidates should know the bitwise and logical operators for their language, and should be able to use them for simple things like setting or testing a specific bit, or set of bits.

Candidates should know about the bit-shift operators in their language, and should know why you would want to use them.

A good weeder question for this area is:

Tell me how to test whether the high-order bit is set in a byte.

Another, more involved one is:

Write a function to count all the bits in an int value; e.g. the function with the signature `int countBits(int x)`

Another good one is:

Describe a function that takes an int value, and returns true if the bit pattern of that int value is the same if you reverse it (i.e. it's a palindrome); i.e. `boolean isPalindrome(int x)`

They don't have to code the last two, just convince you they'd take the right approach. Although if you have them code it correctly, it can count for your Coding weeder question too.

C/C++ programmers should know about the *sizeof* operator and how (and why/when) to use it. Actually, come to think of it, everyone should know this.

All programmers should be able to count in hexadecimal, and should be able to convert between the binary, octal, and hex representations of a number.

Special Fast-Track Version

That's it for the Five Essential Phone Screen Questions. Hope ya liked it.

As a special reward for reading this far, here's a special Bonus Feature: a set of all-too-common answers that are *almost always* indicators of certain failure during our interviews. Even if I'm not on the loop!

Bad Sign #1:

Me: So! What languages have you used, starting with your strongest?
 Them: *(briskly)* C, C++.
 Me: *(long, pregnant pause)*
 Them: *(waiting patiently for me to continue)*
 Me: Any others?
 Them: Nope. C, C++.

Translation: (in thick Southern drawl) "We got both kinds of music here: country *and* western."

Probable failure modes for this candidate:

Will fail the HTML-phone-number question and the OO design question (but will get the OO terminology definitions mostly right.)

Bad Sign #1a:

Me: So! What languages are you most familiar/proficient with?
 Them: *(worried)* I've done mostly Java lately.
 Me: *(long, pregnant pause)*
 Them: Yeah, um, Java.
 Me: Any others?
 Them: Um, I did C in school a long time ago, but... pretty much mostly Java now.

Translation: "Country and Western were both too hardcore for me. I got beat up in a bar."

Probable failure modes for this candidate: Will fail the bits and bytes questions, the HTML-phone-number question, and most of the data structures questions.

Bad Sign #2:

Me: So! What data structures do we have available to us, as programmers?
 Them: Arrays, queues, vectors, stacks, lists, um, linked lists...
 Me: OK, any others?
 Them: Um, doubly-linked lists, and, uh, array lists.
 Me: Have you ever used a tree?
 Them: Oh! *(laughs)* Yeah, um, I forgot about those.

Translation: "My family tree doesn't branch."

Probable failure modes for this candidate: Very likely to fail data structures questions. Will fail any recursive problem, even a simple one like printing the elements of a linked list recursively. Will fail the HTML-phone-number question, since they obviously haven't ever used Perl if "hash" didn't leap to mind.

Bad Sign #3:

Me: So! What the the primitive types in Java (or C++)?
 Them: Ummmm, there's, um, int. And, uh, double.

Me: Any others?

Them: Shoot, I'm drawing a blank right now. Um, String?

Translation: "C made my head hurt. Java is like sweet, sweet aspirin."

Probable failure modes for this candidate: Will fail bits and bytes questions, and probably just about everything else as well.

Bad Sign #4:

Me: So! What text-editor do you use?

Them: Visual Studio.

Me: OK. What about on Unix?

Them: On Unix I use vi.

Me: Er, yeah, vi is cool... ever used VIM?

Them: No, just vi. Always worked just fine for me.

Translation: "Sometimes I type with my elbows when my hands are tired. It's just as fast."

Probable failure modes for this candidate: Will likely fail the HTML-phone-number question. Might pass the interviews, but will need to be scheduled in geologic eras.

Bad Sign #5:

Me: So! What did you study in your Operating Systems class?

Them: Oh, that was a long time ago. I can hardly remember. Hehe.

Me: How long ago was it?

Them: 2 years.

Translation: "I want to use my MBA skills in a dynamic management role. When's lunch?"

Probable failure modes for this candidate: Will probably fail the coding question. Probably any OS questions, too.

All my Insta-Bad Signs above are cliches, in that I've heard these answers from at least 10 to 15 candidates (per question!), none of whom ever got an offer from us. I tend to ask questions like these as a matter of course now.

Summary

This stuff is the ABC's for programmers. Actually it's only going up through maybe J or K; it's not even halfway through the alphabet. But most programmers out there in the Big Wide World will fail utterly in at least one of these areas.

Please cover all five areas if you're a phone screener. If you're the second screener, ask if you don't see evidence of them in the first screener's notes. (And then follow up and remind the first screener they should have asked these things.)

(Published Sep 28th 2004)

Comments

You can put a spin on your 'reverse a string' coding question - first have them write a func that prints out a C string without looping constructs or using local vars. Then if they get that, ask them to implement a reverse string function in the same manner as the first one. Don't say "use recursion" - let them figure out its straightforward applicability to the problem. That's, IMHO, how you can gauge if they 'think recursively' when lightly nudged in that direction:

```
void print(char *s) {
    if (*s != 0) {
        putchar(*s);
        print(s+1);
    }
}
```

```
void printreverse(char *s) {
    if (*s != 0)
        printreverse(s+1);
}
```

```
    putchar(*s);
}
}

int main() {
    char *s = "Hello world";
    print(s);
    putchar('\n');
    printreverse(s);
    putchar('\n');
}
```

Posted by: Martin N. at September 29, 2004 07:39 AM

Nice post Steve - thanks for the checklist.

Would you really accept this answer to the 'Write function to compute Nth fibonacci number' question?

```
static long fib(int n) {
    return n <= 1 ? n : fib(n-1) + fib(n-2);
}
```

I'd hope that most candidates would know that (without memoization of results) the naive recursive solution is $O(n!)$ in time. If they made that error in production code it would be an utter disaster (probably large enough to be noticed early by QA, but still...).

If the candidate didn't at least mention this caveat about their solution, I'd prompt them to compare & contrast with alternatives. If they didn't immediately give the iterative solution and explain the big-O difference, that would be a red flag for me.

regards,

Chris

Posted by: Chris N. at September 29, 2004 08:36 PM

Er, a small error in my comment:

>>>I'd hope that most candidates would know that (without memoization of results) the naive recursive solution is $O(n!)$ in time.

I meant "is $O(2^n)$ in time" of course.

Chris (blushing)

Posted by: Chris N. at September 29, 2004 08:39 PM

Yeah, that factorial fibo solution sucks. I'd be very happy if the candidate told me that they could do it tail-recursively with an accumulator parameter, even if I'm not sure you can do a doubly-recursive call tail-recursively. I'd *still* be happy.

I was thinking of splitting out recursion from basic coding, but that would be Six Essential Areas, and I only have five fingers.

Posted by: Steve Yegge at September 30, 2004 03:42 AM

Interesting post overall...a couple of comments/issues I've seen when doing phone screens:

1. How do you have a candidate read code to you over the phone when the candidate isn't a native English speaker and the phone connection is sub-optimal [I've had this more times than I can remember...]
2. I have some issues with the "scripting" category--it expresses a preference for hacky programmers who prefer speed over maintainability. On our team (Customer Behavior) we're still cleaning up a fair amount of such code that broke the moment a database machine got moved

between data centers. Yes, the code was written quickly, but now our managers are wondering why we can't get to a stable production system so quickly. The section also prefers UNIX programmers over, say, people who worked with Windows for an entire career.

Posted by: Dan at October 7, 2004 01:44 AM

> 1. How do you have a candidate read code to you over the phone
> when the candidate isn't a native English speaker and the phone
> connection is sub-optimal [I've had this more times than I can
> remember...]

Me too.

The best approach I've seen is to give the candidate a "homework" question. Give them an hour to code up a solution to some problem and email it to you. Several teams are doing this regularly, with good results.

> 2. I have some issues with the "scripting" category--it
> expresses a preference for hacky programmers who prefer speed
> over maintainability.

I'm sorry if I gave that impression. We don't want those kinds of programmers here, obviously. The five question areas here are a delicate balance. This category is geared towards determining whether someone has the self-sufficiency to be able to respond quickly to emergencies affecting our customers. They still need to have good judgement and good design skills.

I never actually specify that they need to write it as a script. I just give them problems that are best handled that way: emergency queries and backfills, for example. I've had a few folks burn through 200-line Java or C++ apps that solved the problem, right there on the board, and they got offers.

But after 4 1/2 years over in Customer Service, I've found that knowing how to use "grep" and its ilk is a pretty important survival skill.

And we -are- Unix shop, after all. Unix isn't exactly a niche operating system. There are people who've figured out the basics on their own, even if their professional experience has all been with Microsoft technologies.

But feel free to ask whatever works for you!

Posted by: Steve Yegge at October 7, 2004 02:27 AM

The ultra-cool solution to Fib is the closed-form constant time solution. You need floating point and exponentiation, but that's constant time on modern hardware.

There's also an off-by-one error of sorts in your printOdds() functions. i should start at 1, not 0, or the function should be renamed printEvens().

Posted by: Darren V. at October 7, 2004 02:42 AM

Thanks Darren. Obviously extra-cool solutions are bonus points for the candidate.

Fixed the loop typo. It was caused, interestingly, when I corrected my original working code, after someone emailed me and complained about the performance. Originally it did this:

```
for (int i = 0; i < 100; i++) {  
    if (i % 2 != 0) System.out.println(i);  
}
```

The point was to see if the candidate could write something that worked at all, and I was whipping up examples at 3:00am. Had no idea it would get so much attention.

In any case, someone objected to the fact that it wasn't simply incrementing the loop counter by 2, so it performed poorly. So I went and "optimized" it (an optimization that would go undetected by humans or profilers in this case, as it's totally I/O bound), and in my haste, broke it.

I suppose I should claim that I rigged the whole thing as a demonstration of why optimizing stuff that doesn't matter is needlessly risky. Or that I broke it so I could rant about why unit testing is critical, even for your personal blog content.

But really I just made a bad fix. :)

Thanks again.

Posted by: Steve Yegge at October 7, 2004 03:11 AM

Thanks for posting this. I have been trying to improve my interviewing skills, and this looks like a good set of basics for phone screens.

Posted by: Timothy K. at October 11, 2004 11:20 PM

Just an FYI...

I tried using the find-the-phone-numbers question on a candidate yesterday, and she went straight for Java. I was getting ready to slap her down when I noticed that Java 1.4 contains a regular expression engine. With that enhancement, the write-a-complete-program solution becomes more reasonable.

Posted by: Christopher B. at November 17, 2004 09:37 AM

Yep. Most languages these days have (mostly) Perl5 compatible regexp engines. Java's file handling is a bit more cumbersome than it'd be in a scripting language, but not overly so. The problem is more about understanding fundamental pattern-matching tools than it is about scripting or any particular language.

Posted by: Steve Yegge at November 29, 2004 06:52 PM

I'm about to do my first phone screening and this was very helpful.

Maybe I just don't want to admit that I don't have vital information, but I don't know what 2^{16} is. Jeff Bezos' phone number? I don't understand the significance of knowing powers of two off the top of one's head.

Posted by: Jason R. at March 22, 2005 10:58 PM

Jason: there are many domains for which knowing binary counting is useful, if not essential.

One is when you're doing stuff that involves a lot of bit- and byte-manipulation; examples include network protocols, writing binary serialization/marshalling code, and reading or reverse-engineering file formats.

Another is when you're doing any sort of memory or pointer coding (or more to the point, debugging) with C/C++ code. It's a lot easier to stare at hex-dumps if you're good at translating between decimal and hexadecimal (and binary).

Another broad class of problems involves data whose byte representation is especially significant. UTF-8 and Unicode are good examples. If you ever need to do internationalization, it can be a big help to have a crystal-clear understanding of the meaning of each bit in a byte, short, or int/word value. Actually, this may just be another sub-example of the first domain I mentioned, but you get the idea.

Lastly (and most importantly; all the other examples I've listed pale to insignificance in comparison to this one), it's important for algorithm time and space estimation. If you're trying to decide what data structure to use for a given data set, and you know off the top of your head that 2^{16} is about 65,000, 2^{20} is a million, 2^{32} is 4 billion, and 2^{64} is "big enough", then you'll have an easier time with the decision. Looking at it backwards -- if you use a balanced binary tree (or a log-n search algorithm), you can quickly estimate the base-2 logarithm of your data-set size, which gives you the rough number of steps involved in a lookup operation.

If you *don't* have a good feel for the growth rate of powers of 2, then a little old man will save your daughter, and you'll grant him anything you want in your kingdom, and he'll say he just wants one grain of rice on the first square of a chessboard, then 2 in the second, 4 on the third, and so on. And then: you'll be all out of rice.

Fortunately, you can memorize this stuff in less time than it took you to post your comment. :)

Posted by: Steve Yegge at March 23, 2005 04:11 AM

This is an **excellent** resource, Steve.

I've noticed that a lot of the Java-steeped interviewees will use Strings in their string reverse solutions like so:

```
for (int i = s.length - 1; i >= 0; i--) {  
    returnS += s.substring(i, 1);  
}
```

Most will understand that objects are being allocated and discarded per loop, and will use a StringBuffer to make things "more efficient". Interestingly, a couple of folks couldn't explain *_why_* it would be more efficient.

It's helpful to ask how they would design a limited StringBuffer class using just primitive types to try to help dispel library addiction. It can be surprisingly intimidating for some folks, especially when it comes to reallocating arrays. (Ironic, considering the allocation-fest of their original solutions...)

Thus, I recommend it as another coding question -- though just to talk through, not to code over the phone. Alternatively, asking for the reverse() function using just primitives would do the same thing, but without the scariness of dreaded reallocation.

Posted by: Jeremy D. at March 29, 2005 12:54 AM

[Back to Stevey's Drunken Blog Rants™](#)