Interviewing in Ten Minutes

Spencer Tipping

November 24, 2014

Contents

About this guide

Over the past few years I've worked for a lot of startups. (Because I job-hopped, not because I'm important.) For one reason or another, I ended up conducting technical interviews at about half of these companies. Usually we were looking for developers who knew some high-end functional language, so this guide is strongly biased by that – though I think it also applies to more standard startup jobs.

Everything in this guide is my own opinion. It does not reflect the views of any of my employers, past or present.

1 The business protocol

For various reasons, partly tradition and partly expedience, companies tend to have a set of norms that dictate how people work together. From what I've been able to observe, the following commandments encode the business protocol:

1. **Thou shalt respect other people's time like thine own.** This is absolutely crucial. Business is about money, people convert money to time, and if you waste/take an employee's time then you're wasting the company's money. People holding the money to pay your salary want to see a positive return on their investment in you, which is undermined if you then make other people less productive. **Be the person who makes others more productive.**

2. **Thou shalt get things done.** By this I mean, if someone gives you a task it will be done. They don't have to think twice about it. If you hit difficulty, you'll figure it out. If you can prove it's impossible, change the problem to be possible and explain why you did. **Be the person who gives up last.**

3. **Thou shalt work only when it is productive.** Work ≠ productivity. The two are nearly orthogonal, in fact. Understand what is productive at as high a level as you can, and work on that stuff. Busywork is worse than doing nothing because you'll be unavailable but will be getting nothing done. **Be the person who does everything for a reason.**

4. **Thou shalt simplify.** This is a way to respect people's time. Every problem that comes your way should be simpler after it interacts with you. Make things look easy because to you they are easy. **Be the person who finds the best solution.**

5. **Thou shalt be fluent.** This goes two ways: you must communicate flawlessly both with people and with computers. Functionally, a programmer is an information channel that translates between human intent and computational action. You want this channel to be efficient, lossless, and high-capacity. **Be the best conduit of intent from people to machines.**

6. **Thou shalt not bikeshed.** Thou shalt not argue in general, in fact. But if you must, make sure it's about an issue that justifies a conversation costing the company more than $100/hour (nearly every conversation is likely to cost this, and some will cost far more). **Be the person who cuts through the noise and focuses on real problems.**

I'll go into more details about how to convey these things during an interview later on. For now, though, understand that the above values (or similar ones) serve as the conduit by which your skills can be applied to the company's problems. If this conduit is missing or sufficiently damaged, your skills won't matter.

2 The interviewer

Interviewers are not usually sadists. While the case could be made that the job demands a certain degree of malevolence, most of the confrontation during the interview is due to the economics underlying hiring.

Consider the common flow of resume screen, phone screen, and then in-person interviews. There's a progression of risk/cost associated with moving forward at each stage: screening a resume requires minutes, a phone interview requires an hour, and an in-person interview requires several people-hours, possibly with hotel and airfare. In dollar amounts, these could easily represent investments of $10, $100, and $1000. The worst case, a bad hire, costs potentially $10,000 or more.

Given this, the interviewer operates as an investor of the company's assets.[1] Their job is to evaluate risks for the company by making bets on people. Because of this, if you are an interviewee, your job is to sell such a bet to the person interviewing you. Perhaps more to the point, the interviewer is likely to be conservative because they don't get paid commission; so your goal is to be so awesome that they feel safer saying yes than no.

3 The résumé and cover letter

At $100, a phone screen is still fairly cheap. The stakes of a resume and cover letter are not usually very high, nor is there very much signal for the interviewer to work with. However, if you're about optimizing stuff (and what engineer isn't?), there are a few things you can do to communicate at this point:

1. TEX your resume. There's only upside here, and it's a nice positive signal that you know what you're doing. If you don't know TEX, use HTML and export as PDF. If all else fails, write one in plain text and claim that minimalism is a social virtue you take seriously. Don't use Word because it conveys that you have no standards. **Use a spellchecker.**

2. Keep your resume down to one page. This shows that you respect their time. **Use a spellchecker.**

3. Write a purposeful but informal cover letter. Be personable, reasonably enthusiastic, and talk about how the stuff you do for fun is relevant to the company's hard problems. **Use a spellchecker.**

4. Link to your github profile. If you don't have one yet, create one and upload something. If you don't have anything to upload, start solving Project Euler and upload that. **Use a spellchecker.**

[1] Unlike most investors, however, their pay is constant; so they don't have a stake in the outcome of the interview. This creates a slight negative bias, but as far as I know it shouldn't influence their decisions otherwise.

5. Put some code in your cover letter. Nothing elaborate, but just something you thought was cool, or maybe a challenge problem they have on their website, or something. Do this only if it makes sense, but it's a great opportunity if you can. **Use a spellchecker.**

6. Link to a project you worked on that really describes who you are. **Use a spellchecker.**

There are a few things you absolutely cannot do:

1. Talk about salary in any way. It's just too early and it sends the wrong message. It would be like someone asking about your finances before the first date. This is a question you ask when you're considering marriage, not dinner.

2. Lie. If you're called out on it, you're done for and people will hate you. Word might get around to other companies if you're unlucky. If the truth isn't good enough, fix the truth.

3. Be noticeably unprofessional. Startups aren't overwhelmingly formal places, but they're just as vulnerable to harrassment lawsuits as anyone else. You do not want to give anyone reason to believe that you're going to cause legal problems or alienate people.

4. Bring politics or religion to the table. These topics are emotional catalysts with no upside and enormous potential for problems. Brendan Eich, the inventor of Javascript, was forced to resign as CEO of Mozilla because he had made a $1000 contribution against gay marriage six years prior. If you have public political statements on a blog or elsewhere, take them down.

4 The technical phone interview

At this point you have cost the company $110 or so, and they're deciding whether or not to bump that figure by a factor of 10. This is real money, so the burden is on you to convince the interviewer (who may or may not be good at interviewing people) that you're the best bet they'll make this year. Depending on the job you're applying for, there are a few different ways to do that.

In this section I'm going to assume the technical interview is just talking on the phone. The next section is entirely devoted to how to handle a collaborative coding interview.

4.1 Technical literacy

This is an absolute must. At a minimum, you need to be able to answer questions like these with little or no thought:

- Write the FizzBuzz function.
- Reverse an array in place.
- Reverse a linked list in linear time.
- What is the insertion time for a binary tree?
- What is the insertion time for a hashtable (worst, average)?
- How is a priority queue implemented?
- Quicksort vs mergesort, pros/cons?
- What does $O(n^2)$ mean formally?
- What does "atomic" mean?
- What is memoization?
- Why do floating-point computations often use an epsilon?
- How are negative integers represented?
- Write the Fibonacci function without using recursion.
- What's the difference between the stack and the heap?
- Conceptually, what does a garbage collector do?
- Conceptually, how does virtual memory work?
- What happens under the hood when you go to `google.com`?
- Why should you profile before optimizing?
- What is an example use case for a cache?
- How would you do [some simple task] in a UNIX shell?

4.2 Development literacy

This is also important, especially if you're applying for a position that requires experience. You should be able to speak intelligently about things like:

- The bugs you most commonly write, and habits you've developed to avoid them.
- How expensive it is to fix a bug in production.
- Strengths and weaknesses of unit testing.

- The drawbacks of using powerful languages like Lisp. (You can extol their virtues too, but if that's all you do then people are unlikely to take you very seriously.)
- Your preferences about design and documentation, and why you do it this way.
- A project you've worked on that went really well, and why this happened.
- A project you've worked on that was a disaster, and why. Don't try to blame other people; you need one where you messed up and learned something. If you don't have a project like this, think of something really hard, try to solve it, push the failed results to Github, and talk about why that didn't work. It's fine if failure was inevitable; the main thing is to know how to translate it into improvement.

5 Coding interview

People tend to assume there are two outcomes for a coding interview: succeed or fail. This is not technically true, however; a third outcome is *succeed specatularly*, which is what you should do if you have the option, and here's why.

Coding challenge interviewers are usually engineers, and engineers tend to like to have answers to questions so they can move on. Because of this, they'll try to jump to a conclusion about you as soon as they have enough signal to justify it. This conclusion is rarely final, but at least they have something to work with if they don't observe anything else.

The interview question will usually fall into one of two categories, each equally expedient. One is stuff like FizzBuzz, which rapidly detects certain failure modes. The other is something open-ended that gives you some room to be creative. Although these problems can detect failure, they're more often about detecting excellence.

If at any point in the interview process someone decides you're excellent, you have a much higher chance of getting an offer. Picasso was excellent and his paintings are now worth millions, whereas the guy who paints imitation art just like Picasso is broke. Setting a new bar instantly promotes you into the class of people who can't be replaced, and this makes it much easier to justify further investment in interviewing you.

5.1 Coding challenge etiquette

Enjoy the coding challenge. Relish it. This is an opportunity to be asked difficult, interesting questions that are designed to be solvable in less than an hour. It's break from all the mundane aspects of writing production code, and an opportunity to prove to yourself as well as your interviewer that you can solve anything that comes your way.

- Have a language you're really comfortable with. Java is not a bad choice for this because it's straightforward and everyone knows it.
- Unless the problem is trivial, verbally sketch out your thought process. Argue against the approaches you've taken to see if they hold up. Don't write any code until you know what you're doing. The goal of all this is to demonstrate that your thought process is effective at solving problems. Silence followed by perfection also works, but is riskier.
- Make simplifying assumptions as necessary, but be prepared to undo them later. Always mention when you're doing this.
- If the problem is unsolvable as stated, change it minimally to make it solvable and explain why you need to do this.
- If the problem is too vague or ill-specified to even work with, talk with the interviewer to work the problem into something you can solve. Everyone loses if you walk away from the interview without having done anything.
- When writing code, use idioms that reflect a considered perspective. It should be simple, easy to maintain, well-suited to the problem, and generally self-documenting. You should be able to intelligently discuss every aspect of your coding style.
- Never sweat the small stuff. Nobody cares if you know what the deletion method is called on Java's `HashSet` class. Just casually mention that you'd look it up.
- Never get defensive. If you've made a mistake, fix it quickly (or think about it and discuss it) and move on. Any good interviewer will understand that people make mistakes, and some will even set up questions specifically designed to mislead you. This is all part of the interview and it's to see how you take it when things go wrong.

5.2 Quick-fail problems

These are the easiest problems you'll get, so you need to make them look easy. Some classic examples:

1. Write the factorial function.
2. Write FizzBuzz.
3. Calculate the sum of numbers in a linked list.
4. Print the last line of a file (naive solution).

Coding style matters, as well as speed and whether you can write them without any bugs. I'll go through some solutions of varying quality.

5.2.1 FizzBuzz

Bad solutions indicate a deep lack of understanding of some sort. Any competent interviewer will count a solution like this against you:

Listing 1 examples/fizzbuzz-bad.java

```
pulic int fizz_buzz() {         // "pulic" -- eh; "int" ... ????
int count = 0;                  // count of what?; also, indent!
while (true) {                  // forever? why?
int count2 = count              // "count2"?
while (count2 > 0) {            // modulus! modulus!
count2 = count2 - 3;
}
int count3 = count;             // ok, this is just silly
while (count3 > 0) {
count3 = count3 - 5;
}
//check for divisibility
if (count2 == 0) {
if (count3 == 0) {
System.out.println("fizzbuzz");
}
else                            // missing open brace, not good
System.out.println("fizz");
}
else
if count3 == 0 {
System.out.println("buzz")
}
else {
System.out.println(count)
}
}
}
```

This, on the other hand, is a solution I would consider to be nearly perfect:

Listing 2 examples/fizzbuzz-good.java

```
for (int i = 1; i < 100; ++i) {
  final boolean by3 = i % 3 == 0;
  final boolean by5 = i % 5 == 0;
  if (by3 && by5)
    println("fizzbuzz");
  else if (by5)
    println("buzz");
  else if (by3)
    println("fizz");
```

```
10    else
11      println(i);
12  }
```

Some important differences:

- Indentation and good style should be habit for you. Code should have meaning, not just behavior.
- Know the nuances and idioms of your language. Java gives you a modulus operator, so you should use it here. Use the appropriate iteration and decisional constructs.
- Draw attention to stuff that matters. Here, most of the space is taken up by the condition structure, which is parameterized by two variables calculated above. The `for` loop is appropriately minimal.
- Write the solution to resemble the problem. Here, `by3` and `by5` are the two conditions we care about. This makes the `if` stack nearly a verbatim translation of the problem statement. Because of this, it's very easy to look at the code and be confident it does the right thing.
- Write hints into your code (as appropriate). I like to use `final` to indicate invariance, for example. This is a matter of opinion, but anything you can do to make it clear that you're thinking about maintainability will work in your favor.

5.2.2 Sum of linked list

This problem is nice because it has a mediocre solution:

Listing 3 `examples/sumlist-mediocre.java`

```
1  interface OurList {
2    int head();
3    OurList tail();
4  }
5  int sum(OurList list) {
6    if (list == null)
7      return 0;
8    else
9      return list.head() + sum(list.tail());
10 }
```

This is logically correct, terse, and generally well-expressed. But it has two big problems. First, it returns an `int` rather than a `long` or `double`; this is probably a mistake. Second, and more importantly, it requires linear stack space and may cause an overflow. Java does not optimize tail calls, but even if it did the recursive call here is not in tail position. A much better solution uses iteration:

Listing 4 examples/sumlist-good.java

```
interface OurList {
  int head();
  OurList tail();
}
long sum(OurList xs) {
  long total = 0;
  for (; xs != null; xs = xs.tail())
    total += xs.head();
  return total;
}
```

Not only is this more reliable, but it's also quite a bit faster and follows normal Java idioms more closely. It also creates some opportunities to talk about some edge cases. For example, what if the list is dynamically generated and very long? The iterative solution is ideal here because it loses references to cells it's already processed, which allows them to be garbage collected. This means we can theoretically process lists of arbitrary length using constant space.

I also prefer the parameter name `xs` to a word like `list`. `xs`, which is prounced, "exes," seems simpler to me than `list` does because it refers to a series of things called `x`, and `x` is the most generic name for a data value. It also takes up less space, which makes it easier to focus on the algorithm structure.

The `for` loop is a little nonstandard, but the idiom is preserved from Java's usual numeric iteration. This means that although someone unaccustomed to this type of loop might be confused at first, the usual intuition about `for` loops would all apply: we're moving forward through a list until we hit the end, the termination check is the second thing, and the increment is the third.

5.3 Algorithm problems

These usually have a known-best solution, but it's ok not to find it. You should of course try, and brush up on all of the usual algorithm and data structure stuff before a coding interview in case you get a problem like this. Below I go through some example problems. It's been long enough since I've seen these that I don't remember the solution.

By the way, you'll notice the almost complete absence of comments from my example code. Most interview problems are simple enough that comments are unnecessary. When you're practicing questions like this, write your code without comments or blank lines and see how readable you can make it. It's ok if you later decide to use comments, but most really good code won't require them.

5.3.1 Maximum subarray

> *Given a list of integers, find the offset and length of the subarray with the largest sum.*

Initial train of thought It isn't immediately obvious how to do this, so start by talking through a few strategies. One option is to calculate the sum of every subarray and find the maximum of those, but it seems slow. These subarrays are always made from adjacent elements, though, so maybe we can ask something like, "should I expand out by one." In fact, since we're just summing stuff, we can expand out anytime we have something nonnegative and we'll get a larger, or at least not smaller, sum.

If I were to start coding now (which would be a mistake), here's what I would write:

Listing 5 examples/max-subarray1.java

```
// return [offset, length]
int[] subarray(final int[] xs) {
  int nextUntried  = 1;
  int maxSum       = 0;
  int maxSumStart  = -1;
  int maxSumLength = 0;
  for (int i = 1; i < xs.length; i = nextUntried) {
    maxSumStart  = -1;
    maxSumLength = 1;
    while (i + maxSumStart >= 0 && xs[i + maxSumStart] >= 0) {
      --maxSumStart;
      ++maxSumLength;
    }
    i += maxSumStart;
    while (i + maxSumLength < xs.length
           && xs[i + maxSumLength] >= 0)
      ++maxSumLength;
    nextUntried = i + maxSumLength + 1;
  }
  return new int[] {maxSumStart, maxSumLength};
}
```

The code is a little awkward, but it isn't terrible. The problem is with my thinking: it's obvious I don't really understand something important about the problem. The right move is to continue thinking about it.

First course correction Ok, so what happens when we're expanding and hit a negative number? I guess we need to look on the other side to see if there's an even larger positive number. That's kind of gross, and bumps us back into quadratic time; so let's think about it differently. Suppose we first take a running sum of the array; then what are we looking for? Oh! We're just finding the max and min of that. The numbers in between those extremes collectively have the largest sum.

Writing code here is a much less serious mistake, but is still suboptimal. Here's what I'd say, most likely:

Listing 6 examples/max-subarray2.java

```
int[] subarray(int[] xs) {
  // modify xs in place (!)
  for (int i = 0, total = 0; i < xs.length; ++i)
    xs[i] = total += xs[i];

  int max = 0;
  int min = 0;
  for (int i = 1; i < xs.length; ++i) {
    if (xs[i] > xs[max]) max = i;
    if (xs[i] < xs[min]) min = i;
  }
  return new int[] {Math.min(min, max),
                    Math.max(min, max) - Math.min(min, max)};
}
```

This is much better, but it's wrong and still suboptimal. Let's continue thinking:

Second course correction Ok, so how do we do this in code? We could start by summing the array in a loop, then check for max and min. Oh, but what if min happens after max? I guess we really need the largest min/max gap where the max is after the min.

Do we actually need to sum the array in a loop? We never look backwards at stuff we've summed, so really we just need to go through and maintain a running total. This is an optimal solution, so coding now makes sense:

Listing 7 examples/max-subarray3.java

```
int[] subarray(final int[] xs) {
  if (xs.length == 0)
    return null;
  int max      = 0;
  int min      = 0;
  int maxValue = xs[0];
  int minValue = xs[0];
  for (int i = 1, sum = xs[0]; i < xs.length; ++i) {
    sum += xs[i];
    if (sum < minValue) {
      minValue = maxValue = sum;
      min      = max      = i;
    }
    if (sum > maxValue) {
      maxValue = sum;
      max      = i;
    }
  }
```

```
  return new int[] {min, max - min};
}
```

This solution feels a little more complicated than the last one, but it's conceptually more straightforward (since we're not modifying anything) and better-engineered. Just the fact that we're solving it in constant space is a consideration that a good interviewer would appreciate.

5.3.2 Parse simple math

> *Write a function that takes a string containing a simple expression and returns the result. The expression will contain single-digit numbers, +, and *, and should respect precedence.*

Ok, this is obviously a shunting-yard style problem. We could parse it recursive-descent, but that's overkill. All operators are left-associative, so we can evaluate as we go. Actually, we can just have two constant levels: the number on the left of a + and the current product. So our solution is constant-space.

Listing 8 examples/parse-math.java

```
double parsemath(final String expr) {
  double left  = 0;
  double right = 1;
  for (int i = 0; i < expr.length(); ++i) {
    final char c = expr.charAt(i);
    switch (c) {
      case '+':
        left += right;
        right = 1;
        break;
      case '*':
        break;
      default:
        right *= c - '0';
        break;
    }
  }
  return left + right;
}
```

I had some doubts about 2/3 of the way through this. I wasn't sure it would be ok to add `right` to `left` at the end, since it might contain a stray 1 from either initialization or the + case. But after some thought I realized it would be ok since we always end after seeing a digit. That digit changes the 1 into something meaningful.

In hindsight, my variable names could be better. `left` and `right` are less descriptive than `sum` and `product`. Also, the first time through I had forgotten

parens after `expr.length`. This isn't terrible, but it isn't great either. In my case I was confusing Java and Javascript, but an interviewer might assume I was bad at Java and confusing strings and arrays.

One nice part of this solution is `c - '0'`. Using the equivalent `c - 0x30` or `c - 48` would be hostile to someone reading your code, whereas `c - '0'` gives someone just enough hints that it's clear you're not up to anything too complicated.

Doing things like character-to-number conversion this way tells the interviewer you know how characters work and have at least some knowledge of ASCII. It's also reminiscent of C-style code, which will usually work in your favor in today's hiring culture of very high-level languages and app development (it shows perspective, assuming you can also do the high-level stuff).

5.3.3 Sorted word index

> *Build the index for a book, sorted by word. Your input is a list of ⟨word, page⟩ tuples, and the list may contain duplicates.*

We'll need some data structures for this. I'll go ahead and write this up the wrong way to illustrate some common mistakes:

Listing 9 `examples/sorted-index-bad.java`

```
List index(List tuples) {
  HashMap resultIndex = new HashMap();
  for (int i = 0; i < tuples.size(); ++i) {
    if (!resultIndex.contains(tuples[i].word))
      resultIndex.add(tuples[i].word, new LinkedList());
    resultIndex.get(tuples[i].word).add(tuples[i].page);
  }
  List sortedResult = new ArrayList();
  Iterator iterator = resultIndex.iterator();
  while (iterator.hasNext()) {
    Tuple indexEntry = iterator.next();
    sortedResult.add(indexEntry);
  }
  quicksort(sortedResult);
  return sortedResult;
}
```

This solution has a lot of problems. For one thing, none of the data structures are parameterized. *Always parameterize your data structures.* There are two reasons you want to do this. First, prior to Java 1.5 (a long time ago), Java didn't even have generics. You might come across as a Luddite who prefers inferior technologies.

But more importantly, writing this stuff down tells the interviewer what you're thinking. A map of <Thing, Thing> doesn't convey your intentions very well; its meaning is then defined by what you do to the map instead of

something intrinsic to it.[2] Whenever possible, use code to document what you're doing. Especially parameters and return types, because this gives the interviewer a quick way to correct any inappropriate assumptions you might be making.

Here are some other problems with the code above:

1. It mixes definition types between interfaces (e.g. `List`) and implementations (e.g. `HashMap`). There are very occasionally reasons to do this, but this isn't one of them.
2. It grossly mishandles `tuples`. Were `tuples` a linked list, the function would be quadratic.
3. `resultIndex` is redundant, misleading, and too long. It takes up a huge amount of space inside the `for` loop.
4. `tuples[i].word` is repeated three times in quick succession. Stash this into a short variable.
5. `sortedResult` is too long.
6. `iterator` is redundant and too long. The ideal name is `it`.
7. The `while` loop spends a whole line defining a variable that's almost as long as the expression it replaces.
8. `quicksort()` is not only dangerous performance-wise, but also not a valid Java function. No Java library function would be called this way because Java requires that it be a method call (unless you `import static`, but your interviewer might not assume you meant that).

The code also has some technical issues that I don't think are important for an interview:

1. `Map`s don't support `.add`; the correct method is .put. Nobody cares. The compiler would catch this immediately. If the interviewer makes a big deal about this, find a company with better priorities or more competent interviewers.
2. `tuples[i]` doesn't work. This is fine; you clearly want to get-by-position. The only thing the interviewer should care about here is that this retrieval is linear-time for a `LinkedList`, and you should know this cold and have a very good excuse for doing it this way.
3. `.size()` could be expensive, but isn't. Your interviewer may not have read the source for `LinkedList`, but there's a counter in there that makes `.size()` a constant-time operation. It's cool if you can work this into an interview, but odds are that if you're counting a linked list you're already doing something wrong.

[2] Generics aren't intrinsic really, but you get the idea.

Here's the right way, with some notational liberties that are appropriate during an interview:

Listing 10 examples/sorted-index-good.java

```
SortedMap<String, List<Integer>> index(final List<Tuple> tuples) {
  final SortedMap<...> result = new TreeMap<>();
  for (final Tuple t : tuples) {
    if (result.containsKey(t.word))
      result.get(t.word).add(t.page);
    else
      result.put(t.word, new ArrayList<Integer>(t.page));
  }
  return result;
}
```

`SortedMap<...>` isn't allowed, but it's really obvious what I mean, especially if I explain it as I'm typing. The same thing is true of `new ArrayList<Integer>(t.page)`, which actually is valid Java but doesn't mean what I want it to (in real life it sets the initial capacity of the list). In an interview I would either make a point of mentioning this redefinition, or would just write out the right code.