BBM Interview #1 Study Guide

This document is meant as a guide to the questions we asked in our first interview. Several of these questions are quite challenging, so don't worry if you didn't get them all perfect, or if there are things in this guide you didn't know. Even experienced engineers usually will not get full credit on these questions! What's important is to learn as much as you can from these questions so you can do better in the next practice interview, and then on interviews for jobs.

Most of these questions have a base question, and then multiple follow-ups if there's enough time. It may be that you didn't see any or all of these follow-ups. That's ok too! If you didn't see a particular follow-up, it might be good practice to consider how you'd answer it before looking at the answer. It's also possible that these interview questions change slightly over time, and that this guide doesn't exactly match the questions in your interview.

The implementations in this guide will be in Python, but the thinking process should be applicable to any language. It might be good practice to convert the code from Python to the language of your choice as well.

Finally, this guide is quite long; it's not a book, there's not a plot, you don't have to read all of it in order. If there were particular problems you were interested in or confused by, just read those parts.

With all that said, let's dig in!

Question 1

```
We're writing a new computer game, and we have a list of player scores. We'd like
to figure out how many scores exceed a goal score we have in mind.

Write a function that takes in a list of player scores and a goal, and returns the
number of scores equal to or above our goal

Examples:

scores = [1500, 1600, 1200, 700, 1900, 1300, 1400]
meets_goal(scores, 1450)  => prints 3 (1500, 1600, and 1900 are greater)
```

This question is a warm-up to write some code and get into the mindset of doing coding. While it is a warm-up, we still want to do it correctly, and show as much of our skill as possible.

When presented with a problem, it's often a good idea to think of questions you might ask the interviewer to make sure you understand the question. A common way of losing time in interviews is to answer and write code for the wrong question, so it can be worth it to spend a few seconds asking clarifying questions. Don't spend too long, though! Interviewers often have multiple questions they want to ask, and spending a long time on the first question can hurt your chances later.

Here are a few sample questions you might consider:

- If a score is equal to our goal score, we count it, is that correct?
- What should we do if there are no scores?
- Is there a particular range for the scores or goal?

Another nice technique for harder questions is to come up with a sample test case and see if the interviewer agrees with your answer for it. For example:

```
[0, 100, 200] with a goal of 100 -> 2
```

Let's move on to implementation. The question asks us to write a function that takes in a list of scores and a goal, and outputs the number of scores equal to or greater than the goal. To do that, we'll write a loop that iterates over the scores and counts scores greater than the goal score.

Here's one way to do it.

```
def meets_goal(scores, goal):    # Takes in the scores and the goal as input
    score_count = 0              # Keep count of the scores we've found so far
    for i in range(len(scores)): # Loop from 0 to (length of scores - 1)
                                   # "range" is not inclusive, giving us the -1
        if scores[i] >= goal:    # Check if the i-th score meets our goal
            score_count += 1     # Add 1 to the score_count if so
    return score_count           # Return our answer!

scores = [1500, 1600, 1200, 700, 1900, 1300, 1400]

print(meets_goal(scores, 1450))  # Prints 3
```

However, most languages have a way to loop and get the scores directly. Since we don't need the array index in this case, we can use that method instead to make our code a bit easier to read. (In other languages, look for something like a "for each loop")

```
def meets_goal(scores, goal):
    score_count = 0
    for score in scores:
        if score >= goal:
            score_count += 1
    return score_count
```

If you're really familiar with Python, you could also do something like this:

```
def meets_goal(scores, goal):
    return sum(1 for score in scores if score >= goal)
```

Whichever way you do it, this process of looping over some collection (like a list or tuple) and doing something with each item is quite common, so you should be prepared to do it quickly.

Question 1 Follow-up 1

```
What are some test cases we might create to make sure this function works?
```

Writing test cases is about finding things that could break your code, or more accurately, could break the code of any person who is trying to write a function or tool. It's often better to not think too much about *your* code, and to think more about code in general. This can be difficult, and it comes a lot more easily with practice.

Here are a few things that often make good test cases:

- Boundary conditions. Whenever you have a particular number or place where you're doing something different, it's good to probe around that boundary. Here, we have the "goal", so we might include test cases for:
  - A score right at the goal
  - A score just above the goal
  - A score just below the goal
- "Special" numbers like 0 or negative numbers. Lots of code breaks when given a 0 or a negative number, and the problem did not say scores couldn't be negative. We might consider making tests for:
  - A goal that is 0
  - A goal that is negative
  - A score that is 0

    - A score that is negative
- Things that are very small or very large. For example, here we take in a list of scores. What if that list is very small or very large? Some test cases might be:
    - A list with only one score
    - A list with no scores
    - A list with many scores (although this may be difficult to do in an interview setting)
- Tests that exercise error conditions. This is not as applicable in this question, but if you're writing code that can throw an error, you should write a test to see if it does. For example, if you're opening a file, what happens if the file is not there? Or if you aren't allowed to read it?
- Tests that exercise all reasonable code paths, and expected returns. For example, if you're writing a function that returns whether something is True or False, make sure you write code that returns both True and False (and tests for errors both ways as well)
    - Here, you could write a test that returns 0, as well as a small range of other answers.

Writing good test cases is tricky, and is a skill all of its own that needs development. The more code you write, the more of these things you'll think of, since you'll have made mistakes in your previous code.

Question 1 Follow-up 2

```
Here are some test cases for your code. Try them, and fix your code for any that
give the incorrect result

scores_2 = [-100, -200, -300]
scores_3 = []
scores_4 = [100, 200, 300, 300, 200]
print meets_goal(scores_2, -150)  # prints 1
print meets_goal(scores_3, 300)   # prints 0
print meets_goal(scores_4, 300)   # prints 2
print meets_goal(scores_4, 200)   # prints 4
```

This question just asks you to run the code you wrote on some of our test cases. If your code runs on all of them the first time, great!

If it doesn't, you may want to be careful. When a piece of code you've written fails a test case, it's tempting to add some code to handle that particular case. Sometimes, however, it's better to think about what your code is doing wrong in the first place.

For example, let's say your original code looked like this:

```
def meets_goal(scores, goal):
    score_count = 0
    for score in scores:
        if score > goal:      # This is incorrect!
            score_count += 1
    return score_count
```

Now, when we run the code on these test cases, we get the wrong answer; this prints 1, 0, 0, and 2 instead of 1, 0, 2, and 4. Once we figure out that the problem is scores that are equal to the goal, the correct fix would be to change > to >= in the `if score > goal:` line, but some people are tempted to add code to handle the situation specially like this:

```
def meets_goal(scores, goal):
    score_count = 0
    for score in scores:
        if score > goal:
            score_count += 1
    for score in scores:
        if score == goal:
            score_count += 1
    return score_count
```

It's not as tempting in this problem, but in general when you find out your code is not behaving how you want, adding code is not always the best solution. Try to get to the root of what's going wrong in your code.

Question 1 Follow-up 3

```
Can you make your solution more efficient?
```

The solutions we've seen so far are optimal solutions in terms of space and time complexity. They use O(n) time and O(1) space, where n is the number of items in the scores collection. However, it's useful to think about how we might improve sub-optimal solutions. Let's look at a couple. For each of these, before reading on, think about what the space and time complexity of the code is, and what's making it inefficient.

```
def meets_goal(scores, goal):
    passing_scores = []
    for score in scores:
        if score >= goal": passing_scores.append(score)
    return len(passing_scores)
```

This code still takes O(n) time, but now it also takes O(n) space, due to the creation of the passing_scores list. When you're trying to optimize space complexity, it's useful to look at the data structures you create and think about how you might solve the problem without creating them. Here, we don't need to actually store the passing scores, we can just count them.

```
def meets_goal(scores, goal):
    scores_copy = sorted(scores, reverse=True)
    for i, score in enumerate(scores_copy):
        if score < goal: return i
```

In this solution, the input is sorted from highest score to lowest, and then we look through the input looking for the first score that is lower than the goal. We then return that index (since it says how many scores are greater than that index).

This solution takes O(n log n) time due to the sort. If we were trying to improve this solution, we would try to find ways to do it without the sort, since no sorted solution can run in better than O(n log n) time. However, sometimes you can use a sort to make a more efficient solution; for example, if you have an algorithm that runs in O(n^2) time, you might be able to improve it with a sort.

By the way, the code above has a bug. Can you find it?[1]

Question 1 Follow-up 4

```
Now, instead, we'd like to find the top 3 scores. How could we do that?

scores = [1500, 1600, 1200, 700, 1900, 1300, 1400]
print high_scores_best_three(scores)  # prints 1900, 1600, 1500
```

It's getting trickier now.

A simple way is to sort the scores first and then return the top three like this:

```
def meets_goal(scores, goal):
    return sorted(scores, reverse=True)[:3]
```

This takes O(n log n) time because of the sort. If we use sorted() like this, it also takes O(n) space; if we're allowed to use sort() on the original scores() collection, it takes O(1) space instead.

[1] The code returns None if all of the scores are greater than the goal. We need to add a line at the end "return len(scores)"

How can we do this better? As we talked about in the previous followup, we want to avoid the sort if we can. We're aiming for something that we can do in some number of constant passes through the array.

One way is to use a selection algorithm which you might learn about in an advanced algorithms class. This algorithm can find the 3rd largest number in O(n) time, and then depending on the version of the algorithm you use, you could return all the numbers in the partition greater than that 3rd largest number, or you could scan the list again looking for larger numbers.

Another way would be to scan the array looking for the largest number, and then mark it off in some way. For example, let's say you have the sample input:

```
[1500, 1600, 1200, 700, 1900, 1300, 1400]
```

You would write code looking for the largest number. You'd find the 1900, and put that into your result, then mark it in the list some way so you don't see it again.

```
[1500, 1600, 1200, 700, None, 1300, 1400]
```

You'd repeat this three times to get the three largest numbers. This takes O(kn) time, where k is how many top scores you're looking for. It's worse than using a selection algorithm, but if n is large and k is small, it is still better.

Another way would be to write code like this:

```
def meets_goal(scores, goal):
    best_scores = []
    for score in scores:
        best_scores.append(score)
        best_scores.sort(reverse=True)
        best_scores = best_scores[:3]
    return best_scores
```

This maintains a list of the three largest scores seen so far while iterating through the array. It looks like this would also be O(n log n) due to the sorting, but each time we sort, we're only sorting at most k+1 elements. This algorithm ends up taking O(k log k * n) time.

For questions like this, there may be some clever algorithm that you're not familiar with, but by looking at the code and identifying the part that takes the most time, you may be able to find ways to improve it. And the more questions you practice and review, the more you'll recognize these patterns in interviews.

As a final note on this question, the first solution that sorts the array may have the worst time complexity, but it is also the simplest, and depending on your use case may actually run the fastest. Selection algorithms are quite tricky to implement, and are not actually faster than just sorting until you have millions or sometimes billions of things to sort. Especially in an interview setting, sometimes it's better to implement something simpler, then describe how you would implement something more complex (and only implement that if asked).

Question 1 Follow-up 5

```
Our program has gone into a worldwide release, and we've gotten data from many
players in the form of files containing their scores. We'd like to find the top 100
scores of all time, but we have millions of files with tens of billions of scores.
How might we do this?
```

Most students will never even see this question in the interview, much less solve it. It is really to test the very high-end of candidates.

The key to this problem is that it is unlikely that we could fit tens of billions of scores in memory to do a full sort, or run a full selection algorithm. If each score takes 4 bytes in memory, and we have ten billion, that would require at least 40 billion bytes (around 40 GB) of memory. Some very powerful computers might have that, but if we have "tens" of billions, it isn't feasible.

We won't go in depth on how to implement this, but generally the idea would be to get the top 100 from each file as you read it, and then merge the results file-by-file. This is a limited version of an **external sort**. With many many files, you might like to try to do this in parallel, and could use a tool that implements **MapReduce** like Hadoop.

Question 2

```
We are writing software for a forum about programming interviews, and we would like
to disallow certain words on our forum. Write a function that takes in a string
(representing a forum post) and a list of disallowed words, and changes the
disallowed words to question marks, one for each character in the disallowed word.

Example:
Post: "this binary search tree is perfectly balanced"
Disallowed words: ["tree", "perfectly"]
Result: "this binary search ???? is ????????? balanced"

Note: Time and Space complexity will not be evaluated for this problem.
```

For this problem, while we didn't ask you to implement the problem, it may be a useful exercise to try implementing it on your own as well. It is surprisingly tricky to get just right.

What do we need to do to solve this problem? There are two main parts we have to figure out:

- How to find a disallowed word in a post
- How to replace the word with ?s

To begin with, we'll deal with the most basic version of the problem, where we don't consider any edge cases or complications. First, how can we find a word in a string? Most languages have a method to find a substring within a string, in Python there are two main methods, find() and index(). These both return the starting index where the string is found. The difference between the two is that find() will return -1 if the string isn't found, and index will raise a ValueError. Not raising an error can be useful, but we will have to be careful to check for the -1 value.

```
>>> s = "this is a string"
>>> s.find("is")
2
>>> s.index("is")
2
>>> s.find("test")
-1
>>> s.index("test")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Once we know where the word occurs, we can replace it in a few ways, again depending on language. Since we're not worried about time or space complexity here, one way would be to take the string and split it into three parts, the part before the replacement, the replacement, and the part after the replacement, with techniques like this:

```
>>> s = "this is a string with a bad word"
>>> pos = s.find("bad")
>>> s[:pos]
'this is a string with a '
>>> s[pos:pos+len("bad")]
'bad'
>>> s[pos+len("bad"):]
' word'
>>> s[:pos] + ("?" * len("bad")) + s[pos+len("bad"):]
'this is a string with a ??? word'
```

Another technique might be to use regular expressions. In general, regular expressions can be very powerful for text matching and manipulation, but unless you have a lot of practice they can be difficult to implement correctly, and it's very easy to get them wrong in the pressure of an interview. We don't particularly recommend it.

Some languages also have a replace() method for strings, like this:

```
>>> s = "this is a string with a bad word"
>>> s.replace("bad", "???")
'this is a string with a ??? word'
```

This will work well for now, but it will cause some problems with edge cases in a moment.

Whichever method you use, keep in mind possible edge cases, which brings us to…

Question 2 Follow-up 1

```
Here are a few additional test cases. How would you modify your approach to handle
these?

Post: "A Trie is a great data structure for handling prefix problems!"
Disallowed words: ["trie"]
Result: "A ???? is a great data structure for handling prefix problems!"

Post: "I hate hate hate it when I have an off-by-one error."
Disallowed words: ["hate"]
Result: "I ???? ???? ???? it when I have an off-by-one error"

Post: "I hatehatehate it when I have an off-by-one error."
Disallowed words: ["hate"]
Result: "I ???????????? it when I have an off-by-one error"

Post: "I hate! it when I have an off-by-one error."
Disallowed words: ["hate"]
Result: "I ????! it when I have an off-by-one error"

Post: "I hate it when I have an off-by-one error, don't you?"
Disallowed words: ["hate"]
Result: "I ???? it when I have an off-by-one error, don't you?"

Post: "I absolfreakinglutely love this text editor"
Disallowed words: ["freak"]
Result: "I absol?????inglutely love this text editor"
```

There are a few parts to this question. Primarily, we want to see if we can figure out which of these cases might cause us problems, and why.

Going from top to bottom:

- The first test case will cause problems because our current approaches don't handle words in a case-insensitive way. This is very common in string manipulation problems. A common way to handle this is to match against an upper-case or lower-case version of the string. For example, you might convert the disallowed word and the string to upper case when you do the find. Note that this is a problem that cannot be easily fixed with replace() since if we convert the text to upper case for the replace, it's difficult to put it back.
- The second test case has the same word in it multiple times. We need to make sure that we check for the disallowed words until we don't find one any longer
- The third case expands on the second, by also having the disallowed words be part of a word instead of the word itself.
- The fourth case also adds punctuation, which can also be a problem in string manipulation problems
- The fifth case has a text which has ? in it, which can mess with some uncommon approaches
- The sixth case expands on the third case, putting the disallowed word in the middle of the text.

Here is a sample implementation that covers these test cases. Note that you were not asked to write code during the interview for this question, this is just for reference / study purposes.

```
def q2(post, disallowed):
  upper_post = post.upper()
  changed_post = post
  for word in disallowed:
    upper_word = word.upper()
    current = upper_post.find(upper_word)
    while current != -1:
      changed_post = (changed_post[:current] + ("?" * len(word)) +
                      changed_post[current+len(word):])
      current = upper_post.find(upper_word, current + 1)
  return changed_post
```

In some interviews, you might be asked to improve the time complexity of this approach; the time complexity here is not very good because we're recreating the string every time we replace a word. How might you improve it?[2]

[2] A couple of ideas include making a list of all the sections that need changing and replacing them all at once, or making a character array and replacing them as you find them in a way that doesn't require remaking the string.

Question 2 Follow-up 2

```
In general, what are some difficulties implementing bad word filters such as this
in practice?
```

If you've ever used an internet forum, some of these issues are likely to be familiar to you:

- It's easy to work around filter lists. You can put spaces between words, or substitute characters that look similar (he11o instead of hello, or even hεllo (see https://en.wikipedia.org/wiki/Letterlike_Symbols)
- There may be mistakes filtering partial words. Check out the https://en.wikipedia.org/wiki/Scunthorpe_problem webpage for some examples.
- There can be mistakes filtering out words in the wrong context. For example, "My name is Dick Jones" becomes "My name is ???? Jones"

This is another good example of making tests and seeing if your code works in the way you intend it. If the intent is to stop people from swearing on an internet forum, it may not be effective.

Question 3

```
We have a list of completion times for levels for a game we're building. We'd like
to figure out the fastest completion times for each level.

We have data for plays of each level that include the level number and time for
that level. You can imagine the data looks like this.

Level ID    Time (s) ...(other data)...
  1            84
  2            54
  4            99
  2            56
  3            78
  1            72
  1            72
 ...(rest of data omitted)...

The data is not in any particular order.

How would you use this data to create a data structure containing the fastest
completion time for each level?
```

This problem is meant to be solved using dictionaries (or hashtables / hashmaps / etc.)

Without a dictionary, one brute force solution would be to go through the data once to see what levels exist in the data (here, 1, 2, 3, and 4). Then for each level, we could go through and see what the best time is for that level in the data.

With a dictionary, we can instead do that in one pass. We'll make a dictionary where the key is the level ID, and the value is the best time. This even works in the case that the level IDs are not consecutive, or not even close to each other; you could imagine, if you knew the range of levels and they were all close to each other, using an array instead. For example, if you knew the levels were numbered 0-99, you could make a list with 100 elements in that and use that instead. But even if the level IDs are spread out, or not numbers, you can use a dictionary.

Here's an example of how this might look in code. Note that in the problem specification, the data format is underspecified: we don't know exactly what it looks like. We've made a mock-up of the data for this sample implementation, but it might be done in a different way in practice (maybe the data is in a bunch of objects with other information, or it's in a database table. The same kind of approach can work regardless). Note that you were not asked to provide code during the interview, this is just for reference / study.

```
def q3(data):
  best_times = {}
  for level, time in data:
    # Using float("inf") as the default means the first time is always better
    if best_times.get(level, float("inf")) > time:
      best_times[level] = time
  return best_times

data = [(1, 84), (2, 54), (4, 99), (2, 56), (3, 78), (1, 72), (1, 72)]
print(q3(data))

(Output: {1: 72, 2: 54, 3: 78, 4: 99})
```

Question 3 Follow-ups 1 & 2

```
Follow-up 1: What are the space and time complexities of your solution?

Follow-up 2: Is there any way we could make this solution more efficient?
```

For the solution we provided above, the time complexity is O(n). We're doing constant work (a hashtable retrieve and a comparison) for each item in the data. The space complexity is also O(n) in the worst case. It's possible that every item in the data is a different level, and in that case we'd use O(n) space for the dictionary. If we know up front the number of levels in the data, it's O(|levels|)

If you described an optimal solution during this interview (using a hashtable), you would not be asked the second follow-up. However, this is not always the case in interviews; often an interviewer will ask if there's a way to improve the solution even if the time can't be improved. Here we know that we can't improve the time and space complexity; since the data is not ordered, we have to look at every piece of data to find the best times. We know the space complexity can't be improved since we have to return at least that much data in our function. If you don't see a way to improve time and space complexity in an interview, say so, and explain your reasoning why, it may not be possible!

Question 3 Follow-up 3

```
Next we would like to figure out the most common time for each level. For example,
given the data we can see above, the most common time for level 1 is 72 seconds

How could we do this?
```

We can break this down into two problems: getting a list of all the times for each level, and then finding the most common times.

An inefficient way of getting the times for each level would be similar to the first method described above; getting a list of all the levels, then making one pass for each level to get a list of times for that level, and finally getting the most common time for each level.

A better way is to modify our dictionary approach. However, instead of storing just the fastest time for each level, we'll store all the times for each level instead. One way of doing this is to have the value for the dictionary be a list of times that we append to. We would end up with a data structure like this for the provided data:

```
1 -> [84, 72, 72]
2 -> [54, 56]
3 -> [78]
4 -> [99]
```

Then, we need some way of figuring out which time is most common for each level. One way to do this is to sort the numbers, and then scan through the sorted list looking for the longest run of the same number. So for example:

```
>>> a = [1, 5, 2, 3, 1, 6, 2, 3, 9, 1]
>>> sorted(a)
[1, 1, 1, 2, 2, 3, 3, 5, 6, 9]
```

You can see that when the numbers are sorted, all of the same numbers are grouped together, and you can see which of those groups is largest. Here, 1 is the most common number.

However, sorting takes O(n log n) time where n is how many times there are in the level, and we can do better. A better way of doing this is to use another dictionary. This time, the key will be the time, and the value will be how often that time occurred. Then, we can look through the dictionary to see which value is highest. For the list above, that might look like this:

```
>>> a = [1, 5, 2, 3, 1, 6, 2, 3, 9, 1]
>>> counts = {}       # initialize the dictionary
>>> for number in a:  # Add 1 to a number's count each time we see it.
...   counts[number] = counts.get(number, 0) + 1
...
>>> print(counts)     # Shows (time: count) for each time.
{1: 3, 5: 1, 2: 2, 3: 2, 6: 1, 9: 1}
>>> print(max(counts, key=counts.get))  # Get the key with the max value
1
```

In Python we can do a little better than this. This use case is quite common, so Python has a Counter object in the collections module. Using it looks like this:

```
>>> from collections import Counter
>>> a = [1, 5, 2, 3, 1, 6, 2, 3, 9, 1]
>>> counts = Counter(a)
>>> counts.most_common(1)
[(1, 3)]
```

This is also known as finding the **mode** of a list. Other languages may have statistics libraries for doing this. Python does have a statistics.mode() function, but it doesn't work for our purposes due to a particular edge case we haven't discussed: What if multiple times occur the same number of times? For example, our level 2 times are [54, 56]. These are both equally common, and we should decide what to do about that. Maybe we return any one of them, or we return both. Regardless, because of this, statistics.mode() will not work for us.

```
>>> a = [1, 5, 2, 3, 1, 6, 2, 3, 9, 1]
>>> import statistics
>>> statistics.mode(a)
1
>>> b = [1, 1, 1, 2, 2, 2]
>>> statistics.mode(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/anaconda3/envs/python375/lib/python3.7/statistics.py", line 506, in mode
    'no unique mode; found %d equally common values' % len(table)
statistics.StatisticsError: no unique mode; found 2 equally common values
```

Question 3 Follow-up 4

```
We've found that our numbers are messed up some by speedrunners, who repeatedly run
levels for lower and lower times, and who generally have very similar times from
run to run.

We'd like to get the most common time for each level, only counting the best time
each user has submitted. Let's assume that our data now has a user ID field as
well:

Level ID    Time (s)    User ID  ...(other data)...
  1            84       198eabcd
  2            54       9a8bd98f
  4            99       1983b9a9
  2            56       ef98abc9
  3            78       01bd0a0e
  1            72       87416bcd
  1            72       87416bcd
 ...(rest of data omitted)...

We can see the two 72s times for level 1 have the same user ID.

How can we do this?
```

This can also be solved by nested dictionaries.

First, we'll have a dictionary where the key is the level ID, and the value is a dictionary. This second dictionary will have the user ID be the key, and the best time for that user be the value.

We'll end up with a data structure that looks like this:

```
1 -> 198eabcd -> 84
     87416bcd -> 72
2 -> 9a8bd98f -> 54
     ef98abc9 -> 56
3 -> 01bd0a0e -> 78
4 -> 1983b9a9 -> 99
```

From there, for each level in our upper-level dictionary, we can get the times for each user and get the mode as we described above. The code for that might look like this:

```
import collections

def q3f4(data):
    best_times = {}
    # make dictionary of level -> {user -> time}
    for level, time, user in data:
        if level not in best_times:
            best_times[level] = {}
        if user not in best_times[level]:
            best_times[level][user] = time
        else:
            best_times[level][user] = min(time, best_times[level][user])
    # For each level, get the mode of times for all the users
    result_times = {}
    for level in best_times:
        counts = collections.Counter(best_times[level].values())
        mode = counts.most_common(1)[0][0]
        result_times[level] = mode
    return result_times

data = [(1, 84, "198eabcd"), (2, 54, "9a8bd98f"), (4, 99, "1983b9a9"),
        (2, 56, "ef98abc9"), (3, 78, "01bd0a0e"), (1, 72, "87416bcd"),
        (1, 72, "87416bcd")]

print(q3f4(data))  # prints {1: 84, 2: 54, 4: 99, 3: 78}
```

If we didn't confront the idea of there being multiple modes in the previous follow-up, we definitely should think about it here; every time occurs only once in this data! Regardless, this should give you some ideas of how you might use dictionaries in problems.

Question 4

```
Paul Erdős was a mathematician who published over 1,500 academic papers. The
collaborative distance a person is from Erdős is called an "Erdős number".

For example:
Paul Erdős has a number of 0.
Anyone who wrote a paper with Erdős has a number of 1
Anyone who wrote a paper with someone who has a number of 1 (and did not write a
paper with Erdős himself) has a number of 2
and so on.

We have some data of papers and the people who collaborated on these papers and we
want to be able to figure out, given a name, what their Erdős number is. How can we
do that?

Example:
Paper ID    Authors
000231      Erdős, Piranian, Herzog
010221      Herzog, Stewart
142918      Stewart
176103      Chan, Berne, Stewart, Asensio, Cornwell

From just this data:
Erdős has a number of 0
Piranian and Herzog have a number of 1 (through Erdős)
Stewart has a number of 2 (through Herzog)
```

```
Chan, Berne, Asensio, Cornwell have a number of 3 (through Stewart)
```

This is quite a tricky problem, and it can be difficult to find an approach for it. Often in interviews, it's useful to think about what data structures or algorithms you could use to solve the problem. Let's look at this both ways.

Let's take a look at the data we're given. One thought might be to make a data structure with the paper IDs and authors. We might consider, for example, making a dictionary where the key is the paper ID, and the value is a list of authors. The problem is, it doesn't really seem like the paper ID is a useful key. Nothing in the problem is asking us for the paper IDs or indicates how to use them; this probably isn't the best approach.

If we look at the problem more, it seems like what we want is a way of representing people who wrote papers with other people. When you want a data structure connecting pairs in some way, a graph or a tree is a good way to start. How could we use a graph here?

Usually, we'll want the vertices to be the things you're connecting together, and the edges to represent the connections. So here, the vertices will be the authors, and there will be an edge between two authors if they wrote a paper together. Also, if a person A wrote a paper with person B, then the reverse is also always true, so we can have the edges be undirected.

Once we conceptualize this problem as a graph problem, we can figure out which algorithm to use.

Sometimes it's better to figure out an approach in the other order, figure out which algorithm to use and then try to fit the data into the structure you need to apply that algorithm. Here that may be more difficult. If we realize that what we want is the shortest path from Erdős to the target name, we might realize that we can use breadth-first search, and then try to conceptualize this data as a graph, but that may be more difficult to do. However, when you're working through a problem, you can try to approach it both ways, either from useful data structures, or from useful algorithms.

How do we turn this data into a graph?

```
Paper ID    Authors
000231      Erdős, Piranian, Herzog
010221      Herzog, Stewart
142918      Stewart
176103      Chan, Berne, Stewart, Asensio, Cornwell
```

As described above, we want to make a graph where the names are the vertices, and the edges are whether two names have written a paper together. To do that, for each paper, we'll loop through all pairs of names and add them as edges to our graph.

How should we represent the graph? There are usually two main ways of representing a graph, either an adjacency matrix or an adjacency list. Adjacency matrices are usually better when we want to quickly look up a particular pair of vertices, or when the data is very dense. Here, neither of those are true; common graph search algorithms require a list of neighbors, which works better with an adjacency list. Additionally, the graph is not dense; a dense graph would mean most people write papers with most other people, and that's not true. So we'll use an adjacency list.

Let's start by looking at some code to make the graph data structure for this data. Similar to the last two problems, you were not asked to make this code during the interview, this is for study purposes.

```
data = [("000231", ["Erdos", "Piranian", "Herzog"]),
        ("010221", ["Herzog", "Stewart"]),
        ("142918", ["Stewart"]),
        ("176103", ["Chan", "Berne", "Stewart", "Asensio", "Cornwell"])]

import collections
def make_graph(data):
    # Make a dictionary from name -> [people who wrote papers with name]
    # defaultdict makes it so if a key doesn't exist, it is created with
    # a default value
    # We use a set in case the same two people collaborated multiple times
    graph = collections.defaultdict(set)
    for paper_id, names in data:
        for name1 in names:
            for name2 in names:
                if name1 == name2: continue
                graph[name1].add(name2)
                graph[name2].add(name1)
    return graph

graph = make_graph(data)
for name in graph:
    print(name, graph[name])

-----

Output:
Erdos {'Piranian', 'Herzog'}
Piranian {'Erdos', 'Herzog'}
Herzog {'Erdos', 'Piranian', 'Stewart'}
Stewart {'Chan', 'Cornwell', 'Herzog', 'Asensio', 'Berne'}
Chan {'Cornwell', 'Stewart', 'Asensio', 'Berne'}
Berne {'Cornwell', 'Stewart', 'Chan', 'Asensio'}
Asensio {'Cornwell', 'Stewart', 'Chan', 'Berne'}
Cornwell {'Stewart', 'Chan', 'Asensio', 'Berne'}
```

If we look through the data, we can see this is correct.

Now that we have a graph, we have to decide what algorithm we want to use and implement it. As described before, we want to find the shortest path from Erdos to the name, and this suggests using a breadth-first search. Here's the code.

```
def erdos_number(data, target):
    graph = make_graph(data)
    queue = collections.deque()  # Efficient to add/remove from front/back
                                 # Use a queue for BFS
    visited = set()              # Used to avoid duplicate traversal work
    queue.append((0, "Erdos"))
    visited.add("Erdos")
    while queue:
        number, name = queue.popleft()
        if name == target: return number
        for neighbor in graph[name]:
            if neighbor in visited: continue
            queue.append((number + 1, neighbor))
            visited.add(neighbor)
    return -1  # -1 indicates we didn't find the target.

print(erdos_number(data, "Erdos"))
print(erdos_number(data, "Piranian"))
print(erdos_number(data, "Stewart"))
print(erdos_number(data, "Asensio"))
print(erdos_number(data, "Bob"))  # Not in the data

-----

Output:
0
1
2
3
-1
```

Graph problems, particularly breadth-first search and depth-first search, appear quite frequently in industry interviews, particularly at larger tech companies such as Google, Facebook, etc.. If you're aiming for those kinds of jobs, you should be familiar with them.

Question 4 Follow-up 1

```
We're considering making a website to answer questions about people's Erdős
numbers. However, this data set can be quite large; there are ~400,000 authors and
~1.9 million papers. What could we do to speed up the backend so that we can return
results to individual queries more quickly?
```

There are a few possible answers to this question and questions like it.

When you're asked to speed something up, a common thing you should think about is "what's causing it to be slow in the first place?" Here, we're not sure (and we could ask!) but the implication is that the search itself will be slow with a graph of this size. It may not always be clear what's slow, so if it's not clear from the problem this can be a great clarifying question.

What can we do to improve the speed of this? Here are a few ideas:

- Find a better algorithm. Here, for example, you might use something like bidirectional search, which will often reduce the number of nodes in a graph that need to be searched. You might also be able to find some heuristics that work well in practice (maybe you could search authors with many neighbors first).
- You could make a decision to limit the breadth of the search. For example, you could say that if a person's Erdős number is above 4, you'll return an error message of some kind.
- You could pre-calculate answers, either for the entire graph if you have enough time, or for some subset of people if you don't. If you have enough time when the server starts running to run the full BFS starting from Erdős to every other person, you can return results nearly instantly. This can be a great tradeoff.
- Alternately, if you know that most of your queries are going to be from certain people, or that the underlying graph isn't changing frequently, you could cache answers for popular authors.

Things like pre-computing answers to speed up eventual queries, or caching popular queries, also come up in interviews a lot in practice, so they're great tools to have in your toolbox.

Question 5

```
While exploring an ancient tomb for treasure, you've come across a room with
strange symbols on the floor. These symbols tell you which way you need to walk in
order to pass through the room. However, you're not certain whether it's possible
to follow these directions and reach the end of the room.

You will be given a grid of symbols like the following:
```

```
(start here --->)  >   >   >   >   v
                   v   ^   >   ^   v
                   >   ^   ^   >   E

You start on the square in the upper left, and need to reach the square marked "E".
From each square you must follow the direction on the square. The directions are as
follows:

 v : move down
 > : move right
 ^ : move up
 < : move left

In this case, you to reach the end square:

  [>] [>] [>] [>] [v]
   v   ^   >   ^  [v]
   >   ^   ^   >  [E]

However, the symbols may lead you off the edge.

  Right: off edge
   [>] [>] [v]
    v   ^  [>]
    <   >   E

Also, the end may not be at the bottom right.

 [>] [>] [v] [E]  v
  v   ^  [>] [^]  v
  >   ^   ^   >   >

The path is guaranteed not to loop.

Write a function that determines if it is possible to reach the end of the room
from the start position in the upper left.
```

Conceptually, this problem is straightforward: We start in the upper-left, we follow the path until we reach the "E" square, or until we run off the board. Since we're told the path does not loop, we don't need to keep track of where we've been; if this wasn't true, we might need to store the locations we visit in some data structure as well. This comes up in the follow-up.

However, with grid problems like this it can be complicated to keep track of all the individual parts, especially if we try to put everything into a single function. Additionally, there are many places small programming errors can occur in a problem like this. To help us, it's often useful to break things out into functions. Let's write a main function that describes what we need to do by calling other functions, then fill out those functions.

```
def forced_march(floor):
    row = 0     # Start in upper left corner
    column = 0
    while True:
        if off_edge(floor, row, column): return False   # Check if we've left floor
        if floor[row][column] == "E": return True       # Check if we've hit end
        row, column = next_position(floor, row, column) # Go to next square
```

This outlines the basic flow of the problem. We're going to repeatedly check:

- Have we gone off the floor's edge? If so, we need to return False
- Have we reached the end? If so, we need to return True
- Otherwise, we're not done yet, so we need to go to the next square

We've defined two functions here that we need to write, off_edge() and next_position(). Let's start with off_edge()

```
def off_edge(floor, row, column):
    if row < 0: return True
    if column < 0: return True
    if row >= len(floor): return True
    if column >= len(floor[0]): return True
    return False
```

This function checks each of the four directions, and sees if we're still on the board. For example the first check (row < 0) checks if we're "above" the board, and so on.

This code is not very good stylistically, which we'll get into in a moment. But seeing it broken out this way can be helpful. There are a number of small details to pay attention to here:

- The row numbers can go from 0 to the length of the floor matrix, but not equal to the length of the floor matrix. So make sure 0 is ok, but the exact length of the matrix is not. For example, if there are 5 rows, the numbers 0-4 are valid
- Same with the columns. However, if the matrix is rectangular instead of square, you need to check the length of an individual row to find the number of columns
    - This code assumes the matrix is rectangular, and not "ragged". It's possible to have a matrix that doesn't have the same number of columns for each row. You might want to ask your interviewer if this is true; if it is, you would want to check against floor[row] instead of floor[0].

The most common bugs here are off-by-one errors in the checks, so be very careful when writing code like this.

To come back to the stylistic issues, a very common style issue is to have code that looks like:

`if (boolean expression): return (boolean value)`

Usually this is a mistake, since you may be able to remove the if statement entirely. We can do that here like this:

```
def off_edge(floor, row, column):
    return row < 0 or column < 0 or row >= len(floor) or column >= len(floor[0])
```

This is much shorter, but a bit harder to read; this may not be clearly better. We can go one step further in Python, however, by using advanced Python comparisons:

```
def off_edge(floor, row, column):
    return not (0 <= row < len(floor) and 0 <= column < len(floor[0]))
```

You should write whichever of these you find easiest to write in an interview without errors; it's better to be more verbose and easier to read if there's any chance of confusion.

Next, let's talk about the next_position() function.

```
def next_position(floor, row, column):
    square = floor[row][column]
    if square == "^": return row - 1, column
    if square == ">": return row, column + 1
    if square == "v": return row + 1, column
    if square == "<": return row, column - 1
    return None
```

This takes in a row and column position on the floor, and returns the row and column of the next spot we should check on the floor. This is easy in Python, but in some languages like Java or C++ it may be more difficult to return two values. You could do this by making some sort of a container object, but it may be easier to not break this out into a function. We'll show an example of that in a moment.

This code is also not very complex, but there are a lot of little things that can go wrong. In practice, people writing this code tend to make small mistakes in whether to +1 or -1 a row or a column, or make small errors when copying and pasting. If you write code like this and it doesn't function as you want, read through your code carefully, paying attention to those things. Also, it can be useful to add print statements to your code, seeing what the row and column are before and after and seeing if they are what you want.

Another similar mistake is confusing the row and column. Make sure you name those values in a consistent way, and pass them to your functions consistently; many candidates will swap row and column, or worse call them x and y and then swap those, and it can be very difficult to find mistakes like that.

Here's the full code:

```
def off_edge(floor, row, column):
    return not (0 <= row < len(floor) and 0 <= column < len(floor[0]))

def next_position(floor, row, column):
    square = floor[row][column]
    if square == "^": return row - 1, column
    if square == ">": return row, column + 1
    if square == "v": return row + 1, column
    if square == "<": return row, column - 1
    return None

def forced_march(floor):
    row = 0
    column = 0
    while True:
        if off_edge(floor, row, column): return False
        if floor[row][column] == "E": return True
        row, column = next_position(floor, row, column)
```

Note that, although it may be cleaner and easier to read to break things out into functions like this, it is not necessary. In some languages, as we discussed above, breaking some parts of this into functions can be difficult. Here's a version of this code without helper functions.

```
def forced_march(floor):
    row, column = 0, 0
    while True:
        if row < 0 or row >= len(floor) or column < 0 or column >= len(floor[0]):
            return False
        symbol = floor[row][column]
        if symbol == "^":
            row, column = row - 1, column
        elif symbol == "<":
            row, column = row, column - 1
        elif symbol == "v":
            row, column = row + 1, column
        elif symbol == ">":
            row, column = row, column + 1
        elif symbol == "E":
            return True
```

You can see that the code has many of the same parts.

One other advantage of helper functions is that if the requirements of the problem change, it can be easier to adapt your code. Think about this for the follow-up.

Question 5 Follow-up 1

```
Let's say we change the problem such that some of the squares do not have a
direction on them, like this:

. > > > ^
v . . v >
> ^ v > v
v v < v E

In these squares, you can move whatever direction you'd like.

The problem is still whether it is possible to reach the end from the start. How
would you solve this new problem?
```

This is considerably more difficult than the original question. Since we can move in any direction we'd like in "." squares, we need to check all four directions to see if we can reach the end. This presents us two new problems:

- How can we check in all four directions
- How can we avoid loops / checking forever.

For example, take a look at the square marked "X" below

```
. > > > ^
v X . v >
> ^ v > v
v v < v E
```

If we check all four directions, we have to check down. If we go down, we'll immediately come back up, and be in the X square again. If we don't notice this is happening, we might check down again, and do this forever.

Another way this could happen is by going right from the X square. In that "." square, we could go left, and we could repeat this forever as well.

We can treat this as another graph search problem. A common way to solve this kind of repetition is to keep a set of places we've already been, and if we revisit a place we've already been, we can stop that branch of the search; if we can reach the end from this square, we'll find it in a different branch.

Another common tool to use for this kind of search is recursion. We can think of this as breaking up our problem into different sub-problems. From some particular square in the matrix, we can reach the end if we can go in one of the four directions and reach the end. So we'll call our function in each of the four directions and see if any of them are True. Our recursive function answers the question "can we reach the end from the square we're in", and does so by checking if we can reach the end from any of our neighbors.

The code looks like this:

```
def off_edge(floor, row, column):
    return not (0 <= row < len(floor) and 0 <= column < len(floor[0]))

def next_position(floor, row, column):
    square = floor[row][column]
    if square == "^": return row - 1, column
    if square == ">": return row, column + 1
    if square == "v": return row + 1, column
    if square == "<": return row, column - 1
    return None

def unforced_march(floor, row=0, column=0, visited=None):
    if visited is None:
        visited = set()
        visited.add((row, column))  # should be 0, 0
    if off_edge(floor, row, column): return False
    if floor[row][column] == "E": return True
    if floor[row][column] == ".":  # Check all four directions
        if (row+1, column) not in visited:
            visited.add((row+1, column))
            if unforced_march(floor, row+1, column, visited): return True
        if (row-1, column) not in visited:
            visited.add((row-1, column))
            if unforced_march(floor, row-1, column, visited): return True
        if (row, column+1) not in visited:
            visited.add((row, column+1))
            if unforced_march(floor, row, column+1, visited): return True
        if (row, column-1) not in visited:
            visited.add((row, column-1))
            if unforced_march(floor, row, column-1, visited): return True
    else:
        next_row, next_column = next_position(floor, row, column)
        visited.add((next_row, next_column))
        return unforced_march(floor, next_row, next_column, visited)
    return False
```

This works, but if you look at it there's a problem. The code that goes in all four directions looks largely the same, with just small differences in whether we're adding or subtracting from a row or column. We'd like this to be cleaner. We can remove a lot of the duplication like this:

```
def off_edge(floor, row, column):
    return not (0 <= row < len(floor) and 0 <= column < len(floor[0]))

def next_position(floor, row, column):
    square = floor[row][column]
    if square == "^": return row - 1, column
    if square == ">": return row, column + 1
    if square == "v": return row + 1, column
    if square == "<": return row, column - 1
    return None

def unforced_march(floor, row=0, column=0, visited=None):
    if visited is None:
        visited = set()
        visited.add((row, column))  # should be 0, 0
    if off_edge(floor, row, column): return False
    if floor[row][column] == "E": return True
    if floor[row][column] == ".":  # Check all four directions
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for row_change, column_change in directions:
            new_row, new_column = row + row_change, column + column_change
            if (new_row, new_column) not in visited:
                visited.add((new_row, new_column))
                if unforced_march(floor, new_row, new_column, visited): return True
    else:
        next_row, next_column = next_position(floor, row, column)
        visited.add((next_row, next_column))
        return unforced_march(floor, next_row, next_column, visited)
    return False
```

There is still some duplication we would like to remove; a lot of the code in next_position() seems similar to the code in the four direction checks, and we might be able to make some of this cleaner by using a BFS instead of a DFS, or by making the function iterative instead of recursive. However, I'll leave those changes to you. :-)

Conclusion

Taking software engineering interviews, particularly at large companies, can be difficult and stressful. It requires a lot of study and practice to do well at coding interviews. Hopefully, this guide has been enough to show how to approach some of these problems, and some of the things you might run into in live interviews.

The best way to get better is to continue practicing. There are a number of resources online with practice problems of all levels; for some good examples, check out sites like LeetCode, Pramp, or Interview Cake, books like Cracking the Coding Interview, or put "practice interview problems" into your favorite search engine. What is important is to pick a resource that you can get feedback from as you practice. For example, seeing sample solutions, or being told in some way whether your solution is correct or not.

Good luck! If you have any questions, please email practice@karat.com, and come to our Friday workshops and office hours. Hope to see you there!