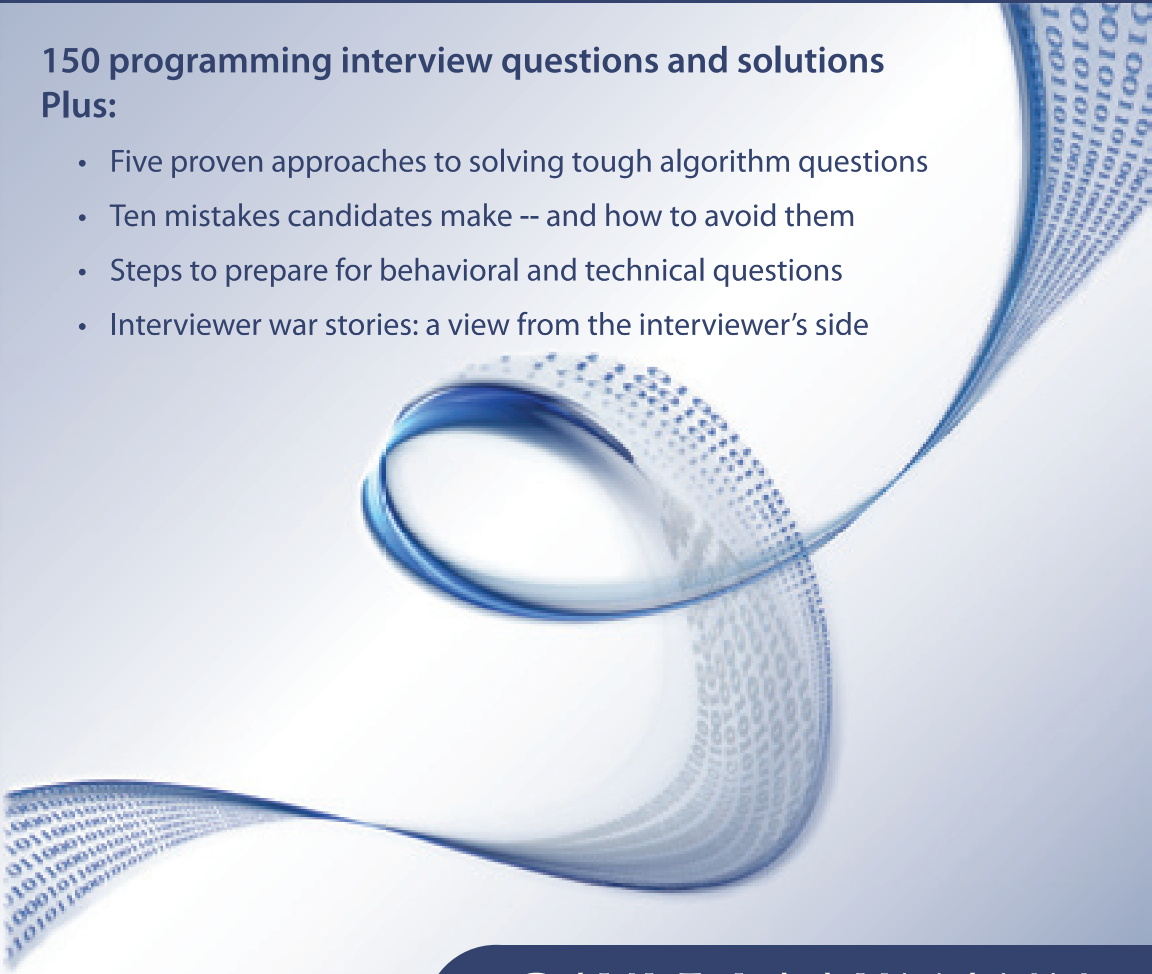
FOURTH EDITION

CRACKING THE CODING INTERVIEW

150 programming interview questions and solutions
Plus:

- Five proven approaches to solving tough algorithm questions
- Ten mistakes candidates make -- and how to avoid them
- Steps to prepare for behavioral and technical questions
- Interviewer war stories: a view from the interviewer's side



GAYLE LAAKMANN
Founder and CEO, CareerCup.com

CRACKING THE CODING INTERVIEW

150 Programming Interview Questions and Solutions

GAYLE LAAKMANN
Founder and CEO, CareerCup.com

CareerCup, LLC
Seattle, WA

CRACKING THE CODING INTERVIEW, FOURTH EDITION

Published by CareerCup, LLC, Seattle, WA. Version 3.21090410302210.

Printed in United States of America

978-1-450-59320-5 9781450593205 (ISBN 13)

General Advice for Technical Questions

Interviews are supposed to be difficult. If you don't get every – or any – answer immediately, that's ok! In fact, in my experience, maybe only 10 people out of the 120+ that I've interviewed have gotten the question right instantly.

So when you get a hard question, don't panic. Just start talking aloud about how you would solve it.

And, one more thing: you're not done until the interviewer says that you're done! What I mean here is that when you come up with an algorithm, start thinking about the problems accompanying it. When you write code, start trying to find bugs. If you're anything like the other 110 candidates that I've interviewed, you probably made some mistakes.

Five Steps to a Technical Questions

A technical interview question can be solved utilizing a five step approach:

1. Ask your interviewer questions to resolve ambiguity.
2. Design an Algorithm
3. Write pseudo-code first, but make sure to tell your interviewer that you're writing pseudo-code! Otherwise, he/she may think that you're never planning to write "real" code, and many interviewers will hold that against you.
4. Write your code, not too slow and not too fast.
5. Test your code and *carefully* fix any mistakes.

Step 1: Ask Questions

Technical problems are more ambiguous than they might appear, so make sure to ask questions to resolve anything that might be unclear or ambiguous. You may eventually wind up with a very different – or much easier – problem than you had initially thought. In fact, many interviewers (especially at Microsoft) will specifically test to see if you ask good questions.

Good questions might be things like: What are the data types? How much data is there? What assumptions do you need to solve the problem? Who is the user?

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

» *Question: What sort of list? An array? A linked list?*

» Answer: An array.

» *Question: What does the array hold? Numbers? Characters? Strings?*

» Answer: Numbers.

» *Question: And are the numbers integers?*

» Answer: Yes.

» *Question: Where did the numbers come from? Are they IDs? Values of something?*

» Answer: They are the ages of customers.

» *Question: And how many customers are there?*

» Answer: About a million.

We now have a pretty different problem: sort an array containing a million integers between 0 and 130. How do we solve this? Just create an array with 130 elements and count the number of ages at each value.

Step 2: Design an Algorithm

Designing an algorithm can be tough, but our five approaches to algorithms can help you out (see pg 34). While you're designing your algorithm, don't forget to think about:

» What are the space and time complexities?

» What happens if there is a lot of data?

» Does your design cause other issues? (i.e., if you're creating a modified version of a binary search tree, did your design impact the time for insert / find / delete?)

» If there are other issues, did you make the right trade-offs?

» If they gave you specific data (e.g., mentioned that the data is ages, or in sorted order), have you leveraged that information? There's probably a reason that you're given it.

Step 3: Pseudo-Code

Writing pseudo-code first can help you outline your thoughts clearly and reduce the number of mistakes you commit. But, make sure to tell your interviewer that you're writing pseudo-code first and that you'll follow it up with "real" code. Many candidates will write pseudo-code in order to 'escape' writing real code, and you certainly don't want to be confused with those candidates.

Step 4: Code

You don't need to rush through your code; in fact, this will most likely hurt you. Just go at a nice, slow methodical pace. Also, remember this advice:

» Use Data Structures Generously: Where relevant, use a good data structure or define your own. For example, if you're asked a problem involving finding the minimum age for a group of people, consider defining a data structure to represent a Person. This shows your interviewer that you care about good object oriented design.

» Don't Crowd Your Coding: This is a minor thing, but it can really help. When you're writing code on a whiteboard, start in the upper left hand corner – not in the middle. This will give you plenty of space to write your answer.

Step 5: Test

Yes, you need to test your code! Consider testing for:

» Extreme cases: 0, negative, null, maximums, etc

» User error: What happens if the user passes in null or a negative value?

» General cases: Test the normal case.

If the algorithm is complicated or highly numerical (bit shifting, arithmetic, etc), consider testing while you're writing the code rather than just at the end.

Also, when you find mistakes (which you will), carefully think through *why* the bug is occuring. One of the worst things I saw while interviewing was candidates who recognized a mistake and tried making "random" changes to fix the error.

For example, imagine a candidate writes a function that returns a number. When he tests his code with the number '5' he notices that it returns 0 when it should be returning 1. So, he changes the last line from "return ans" to "return ans+1," without thinking through why this would resolve the issue. Not only does this look bad, but it also sends the candidate on an endless string of bugs and bug fixes.

When you notice problems in your code, really think deeply about why your code failed before fixing the mistake.

Five Algorithm Approaches

There's no sure fire approach to solving a tricky algorithm problem, but the approaches below can be useful. Keep in mind that the more problems you practice, the easier it will to identify which approach to use.

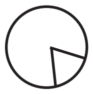
Also, remember that the five approaches can be "mixed and matched." That is, once you've applied "Simplify & Generalize", you may want to implement Pattern Matching next.

APPROACH I: EXAMPLIFY

Description: Write out specific examples of the problem, and see if you can figure out a general rule.

Example: Given a time, calculate the angle between the hour and minute hands.

Approach: Start with an example like 3:27. We can draw a picture of a clock by selecting where the 3 hour hand is and where the 27 minute hand is.



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

» Minute angle (from 12 o'clock): 360 * minutes / 60

» Hour angle (from 12 o'clock): 360 * (hour % 12) / 12 + 360 * (minutes / 60) * (1 / 12)

» Angle between hour and minute: (hour angle - minute angle) % 360

By simple arithmetic, this reduces to 30 * hours - 5.5 * minutes.

APPROACH II: PATTERN MATCHING

Description: Consider what problems the algorithm is similar to, and figure out if you can modify the solution to develop an algorithm for this problem.

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

Similar Problems:

» Find the minimum element in an array.

» Find a particular element in an array (eg, binary search).

Algorithm:

Finding the minimum element in an array isn't a particularly interesting algorithm (you could just iterate through all the elements), nor does it use the information provided (that the array is sorted). It's unlikely to be useful here.

However, binary search is very applicable. You know that the array is sorted, but rotated. So, it must proceed in an increasing order, then reset and increase again. The minimum element is the "reset" point.

If you compare the first and middle element (3 and 6), you know that the range is still increasing. This means that the reset point must be after the 6 (or, 3 is the minimum element and the array was never rotated). We can continue to apply the lessons from binary search to pinpoint this reset point, by looking for ranges where LEFT > RIGHT. That is, for a particular point, if LEFT < RIGHT, then the range does not contain the reset. If LEFT > RIGHT, then it does.

APPROACH III: SIMPLIFY & GENERALIZE

Description: Change a constraint (data type, size, etc) to simplify the problem. Then try to solve it. Once you have an algorithm for the "simplified" problem, generalize the problem again.

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

Simplification: Instead of solving the problem with words, solve it with characters. That is, imagine we are cutting characters out of a magazine to form a ransom note.

Algorithm:

We can solve the simplified ransom note problem with characters by simply creating an array and counting the characters. Each spot in the array corresponds to one letter. First, we count the number of times each character in the ransom note appears, and then we go through the magazine to see if we have all of those characters.

When we generalize the algorithm, we do a very similar thing. This time, rather than creating an array with character counts, we create a hash table. Each word maps to the number of times the word appears.

APPROACH IV: BASE CASE AND BUILD

Description: Solve the algorithm first for a base case (e.g., just one element). Then, try to solve it for elements one and two, assuming that you have the answer for element one. Then, try to solve it for elements one, two and three, assuming that you have the answer to elements one and two.

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

Test String: abcdefg

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

This is the first "interesting" case. If we had the answer to P("ab"), how could we generate P("abc"). Well, the additional letter is "c", so we can just stick c in at every possible point. That is:

```
    merge(c, ab) --> cab, acb, abc
    merge(c, ba) --> cba, bca, bac
```

Algorithm: Use a recursive algorithm. Generate all permutations of a string by "chopping off" the last character and generating all permutations of s[1… n-1]. Then, insert s[n] into every location of the string.

NOTE: Base Case and Build Algorithms often lead to natural recursive algorithms.

APPROACH V: DATA STRUCTURE BRAINSTORM

Description: This is hacky, but it often works. Simply run through a list of data structures and try to apply each one.

Example: Numbers are randomly generated and stored into an (expanding) array. How would you keep track of the median?

Data Structure Brainstorm:

» Linked list? Probably not – linked lists tend not to do very well with accessing and sorting numbers.

» Array? Maybe, but you already have an array. Could you somehow keep the elements sorted? That's probably expensive. Let's hold off on this and return to it if it's needed.

» Binary tree? This is possible, since binary trees do fairly well with ordering. In fact, if the binary search tree is perfectly balanced, the top might be the median. But, be careful – if there's an even number of elements, the median is actually the average of the middle two elements. The middle two elements can't both be at the top. This is probably a workable algorithm, but let's come back to it.

» Heap? A heap is really good at basic ordering and keeping track of max and mins. This is actually interesting – if you had two heaps, you could keep track of the biggest half and the smallest half of the elements. The biggest half is kept in a min heap, such that the smallest element in the biggest half is at the root. The smallest half is kept in a max heap, such that the biggest element of the smallest half is at the root. Now, with these data structures, you have the potential median elements at the roots. If the heaps are no longer the same size, you can quickly "rebalance" the heaps by popping an element off the one heap and pushing it onto the other.

Note that the more problems you do, the better instinct you will develop about which data structure to apply.

Part 1

Data Structures

Chapter 1 | Arrays and Strings

Hash Tables

While not all problems can be solved with hash tables, a shocking number of interview problems can be. Before your interview, make sure to practice both using and implementing hash tables.

```
public HashMap<Integer, Student> buildMap(Student[] students) {
    HashMap<Integer, Student> map = new HashMap<Integer, Student>();
    for (Student s : students) map.put(s.getId(), s);
    return map;
}
```

ArrayList (Dynamically Resizing Array):

An ArrayList, or a dynamically resizing array, is an array that resizes itself as needed while still providing O(1) access. A typical implementation is that when a vector is full, the array doubles in size. Each doubling takes O(n) time, but happens so rarely that its amortized time is still O(1).

```
public ArrayList<String> merge(String[] words, String[] more) {
    ArrayList<String> sentence = new ArrayList<String>();
    for (String w : words) sentence.add(w);
    for (String w : more) sentence.add(w);
    return sentence;
}
```

StringBuffer / StringBuilder

Question: What is the running time of this code?

```
public String makeSentence(String[] words) {
    StringBuffer sentence = new StringBuffer();
    for (String w : words) sentence.append(w);
    return sentence.toString();
}
```

Answer: O(n^2), where n is the number of letters in sentence. Here's why: each time you append a string to sentence, you create a copy of sentence and run through all the letters in sentence to copy them over. If you have to iterate through up to n characters each time in the loop, and you're looping at least n times, that gives you an O(n^2) run time. Ouch!

With StringBuffer (or StringBuilder) can help you avoid this problem.

```
public String makeSentence(String[] words) {
    StringBuffer sentence = new StringBuffer();
    for (String w : words) sentence.append(w);
    return sentence.toString();
}
```

Chapter 1 | Arrays and Strings

1.1 Implement an algorithm to determine if a string has all unique characters. What if you can not use additional data structures?

pg 95

1.2 Write code to reverse a C-Style String. (C-String means that "abcd" is represented as five characters, including the null character.)

pg 96

1.3 Design an algorithm and write code to remove the duplicate characters in a string without using any additional buffer. NOTE: One or two additional variables are fine. An extra copy of the array is not.

FOLLOW UP

Write the test cases for this method.

pg 97

1.4 Write a method to decide if two strings are anagrams or not.

pg 99

1.5 Write a method to replace all spaces in a string with '%20'.

pg 100

1.6 Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?

pg 101

1.7 Write an algorithm such that if an element in an MxN matrix is 0, its entire row and column is set to 0.

pg 102

1.8 Assume you have a method isSubstring which checks if one word is a substring of another. Given two strings, s1 and s2, write code to check if s2 is a rotation of s1 using only one call to isSubstring (i.e., "waterbottle" is a rotation of "erbottlewat").

pg 103

Chapter 2 | Linked Lists

How to Approach:

Linked list questions are extremely common. These can range from simple (delete a node in a linked list) to much more challenging. Either way, we advise you to be extremely comfortable with the easiest questions. Being able to easily manipulate a linked list in the simplest ways will make the tougher linked list questions much less tricky. With that said, we present some "must know" code about linked list manipulation. You should be able to easily write this code yourself prior to your interview.

Creating a Linked List:

NOTE: When you're discussing a linked list in an interview, make sure to understand whether it is a single linked list or a doubly linked list.

```
class Node {
    Node next = null;
    int data;
    public Node(int d) { data = d; }
    void appendToTail(int d) {
        Node end = new Node(d);
        Node n = this;
        while (n.next != null) { n = n.next; }
        n.next = end;
    }
}
```

Deleting a Node from a Singly Linked List

```
Node deleteNode(Node head, int d) {
    Node n = head;
    if (n.data == d) {
        return head.next; /* moved head */
    }
    while (n.next != null) {
        if (n.next.data == d) {
            n.next = n.next.next;
            return head; /* head didn't change */
        }
        n = n.next;
    }
}
```

Chapter 2 | Linked Lists

2.1 Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

pg 105

2.2 Implement an algorithm to find the nth to last element of a singly linked list.

pg 106

2.3 Implement an algorithm to delete a node in the middle of a single linked list, given only access to that node.

EXAMPLE

Input: the node 'c' from the linked list a->b->c->d->e

Result: nothing is returned, but the new linked list looks like a->b->d->e

pg 107

2.4 You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE
Input: (3 -> 1 -> 5) + (5 -> 9 -> 2)

Output: 8 -> 0 -> 8

pg 108

2.5 Given a circular linked list, implement an algorithm which returns node at the beginning of the loop.

DEFINITION
Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

input: A -> B -> C -> D -> E -> C [the same C as earlier]

output: C

pg 109

Chapter 3 | Stacks and Queues

How to Approach:

Whether you are asked to implement a simple stack / queue, or you are asked to implement a modified version of one, you will have a big leg up on other candidates if you can flawlessly work with stacks and queues. Practice makes perfect! Here is some skeleton code for a Stack and Queue class.

Implementing a Stack

```
class Stack {
    Node top;
    Node pop() {
        if (top != null) {
            Object item = top.data;
            top = top.next;
            return item;
        }
        return null;
    }
    void push(Object item) {
        Node t = new Node(item);
        t.next = top;
        top = t;
    }
}
```

Implementing a Queue

```
class Queue {
    Node first, last;
    void enqueue(Object item) {
        if (!first) {
            back = new Node(item);
            first = back;
        } else {
            back.next = new Node(item);
            back = back.next;
        }
    }
    Node dequeue(Node n) {
        if (front != null) {
            Object item = front.data;
            front = front.next;
            return item;
        }
        return null;
    }
}
```

Chapter 3 | Stacks and Queues

3.1 Describe how you could use a single array to implement three stacks.

pg 111

3.2 How would you design a stack which, in addition to push and pop, also has a function min which returns the minimum element? Push, pop and min should all operate in O(1) time.

pg 113

3.3 Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure SetOfStacks that mimics this. SetOfStacks should be composed of several stacks, and should create a new stack once the previous one exceeds capacity. SetOfStacks.push() and SetOfStacks.pop() should behave identically to a single stack (that is, pop() should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function popAt(int index) which performs a pop operation on a specific sub-stack.

pg 115

3.4 In the classic problem of the Towers of Hanoi, you have 3 rods and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (e.g., each disk sits on top of an even larger one). You have the following constraints:

(A) Only one disk can be moved at a time.

(B) A disk is slid off the top of one rod onto the next rod.

(C) A disk can only be placed on top of a larger disk.

Write a program to move the disks from the first rod to the last using Stacks.

pg 118

3.5 Implement a MyQueue class which implements a queue using two stacks.

pg 120

3.6 Write a program to sort a stack in ascending order. You should not make any assumptions about how the stack is implemented. The following are the only functions that should be used to write this program: push | pop | peek | isEmpty.

pg 121