10.2 **Group Anagrams:** Write a method to sort an array of strings so that all the anagrams are next to each other.

Hints: #177, #182, #263, #342

pg 397

10.3 ***Search in Rotated Array:*** Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE

Input: find 5 in `{15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}`

Output: 8 (the index of 5 in the array)

Hints: #298, #310

pg 398

10.4 **Sorted Search, No Size:** You are given an array-like data structure `Listy` which lacks a size method. It does, however, have an `elementAt(i)` method that returns the element at index `i` in `O(1)` time. If `i` is beyond the bounds of the data structure, it returns `-1`. (For this reason, the data structure only supports positive integers.) Given a `Listy` which contains sorted, positive integers, find the index at which an element x occurs. If x occurs multiple times, you may return any index.

Hints: #320, #337, #348

pg 400

10.5 **Sparse Search:** Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

Input: `ball`, `{"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}`

Output: 4

Hints: #256

pg 401

10.6 **Sort Big File:** Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

Hints: #207

pg 402

10.7 **Missing Int:** Given an input file with four billion non-negative integers, provide an algorithm to generate an integer that is not contained in the file. Assume you have 1 GB of memory available for this task.

FOLLOW UP

What if you have only 10 MB of memory? Assume that all the values are distinct and we now have no more than one billion non-negative integers.

Hints: #235, #254, #281

pg 403

10.8 Find Duplicates: You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

Hints: #289, #315

pg 406

10.9 Sorted Matrix Search: Given an `M x N` matrix in which each row and each column is sorted in ascending order, write a method to find an element.

Hints: #193, #211, #229, #251, #266, #279, #288, #291, #303, #317, #330

pg 407

10.10 Rank from Stream: Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

EXAMPLE

Stream (in order of appearance): `5, 1, 4, 4, 5, 9, 7, 13, 3`

```
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1
getRankOfNumber(4) = 3
```

Hints: #301, #376, #392

pg 412

10.11 Peaks and Valleys: In an array of integers, a "peak" is an element which is greater than or equal to the adjacent integers and a "valley" is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

Hints: #196, #219, #231, #253, #277, #292, #316

pg 414

Additional Questions: Arrays and Strings (#1.2), Recursion (#8.3), Moderate (#16.10, #16.16, #16.21, #16.24), Hard (#17.11, #17.26).

Hints start on page 662.

11

Testing

Before you flip past this chapter saying, "but I'm not a tester," stop and think. Testing is an important task for a software engineer, and for this reason, testing questions may come up during your interview. Of course, if you are applying for Testing roles (or Software Engineer in Test), then that's all the more reason why you need to pay attention.

Testing problems usually fall under one of four categories: (1) Test a real world object (like a pen); (2) Test a piece of software; (3) Write test code for a function; (4) Troubleshoot an existing issue. We'll cover approaches for each of these four types.

Remember that all four types require you to not make an assumption that the input or the user will play nice. Expect abuse and plan for it.

▸ What the Interviewer Is Looking For

At their surface, testing questions seem like they're just about coming up with an extensive list of test cases. And to some extent, that's right. You do need to come up with a reasonable list of test cases.

But in addition, interviewers want to test the following:

- *Big Picture Understanding:* Are you a person who understands what the software is really about? Can you prioritize test cases properly? For example, suppose you're asked to test an e-commerce system like Amazon. It's great to make sure that the product images appear in the right place, but it's even more important that payments work reliably, products are added to the shipment queue, and customers are never double charged.
- *Knowing How the Pieces Fit Together:* Do you understand how software works, and how it might fit into a greater ecosystem? Suppose you're asked to test Google Spreadsheets. It's important that you test opening, saving, and editing documents. But, Google Spreadsheets is part of a larger ecosystem. You need to test integration with Gmail, with plug-ins, and with other components.
- *Organization:* Do you approach the problem in a structured manner, or do you just spout off anything that comes to your head? Some candidates, when asked to come up with test cases for a camera, will just state anything and everything that comes to their head. A good candidate will break down the parts into categories like Taking Photos, Image Management, Settings, and so on. This structured approach will also help you to do a more thorough job creating the test cases.
- *Practicality:* Can you actually create reasonable testing plans? For example, if a user reports that the software crashes when they open a specific image, and you just tell them to reinstall the software, that's typically not very practical. Your testing plans need to be feasible and realistic for a company to implement.

Demonstrating these aspects will show that you will be a valuable member of the testing team.

▸ Testing a Real World Object

Some candidates are surprised to be asked questions like how to test a pen. After all, you should be testing software, right? Maybe, but these "real world" questions are still very common. Let's walk through this with an example.

Question: How would you test a paperclip?

Step 1: Who will use it? And why?

You need to discuss with your interviewer who is using the product and for what purpose. The answer may not be what you think. The answer could be "by teachers, to hold papers together," or it could be "by artists, to bend into the shape of animal." Or, it could be both. The answer to this question will shape how you handle the remaining questions.

Step 2: What are the use cases?

It will be useful for you to make a list of the use cases. In this case, the use case might be simply fastening paper together in a non-damaging (to the paper) way.

For other questions, there might be multiple use cases. It might be, for example, that the product needs to be able to send and receive content, or write and erase, and so on.

Step 3: What are the bounds of use?

The bounds of use might mean holding up to thirty sheets of paper in a single usage without permanent damage (e.g., bending), and thirty to fifty sheets with minimal permanent bending.

The bounds also extend to environmental factors as well. For example, should the paperclip work during very warm temperatures (90 - 110 degrees Fahrenheit)? What about extreme cold?

Step 4: What are the stress / failure conditions?

No product is fail-proof, so analyzing failure conditions needs to be part of your testing. A good discussion to have with your interviewer is about when it's acceptable (or even necessary) for the product to fail, and what failure should mean.

For example, if you were testing a laundry machine, you might decide that the machine should be able to handle at least 30 shirts or pants. Loading 30 - 45 pieces of clothing may result in minor failure, such as the clothing being inadequately cleaned. At more than 45 pieces of clothing, extreme failure might be acceptable. However, extreme failure in this case should probably mean the machine never turning on the water. It should certainly *not* mean a flood or a fire.

Step 5: How would you perform the testing?

In some cases, it might also be relevant to discuss the details of performing the testing. For example, if you need to make sure a chair can withstand normal usage for five years, you probably can't actually place it in a home and wait five years. Instead, you'd need to define what "normal" usage is (How many "sits" per year on the seat? What about the armrest?). Then, in addition to doing some manual testing, you would likely want a machine to automate some of the usage.

Testing a Piece of Software

Testing a piece of software is actually very similar to testing a real world object. The major difference is that software testing generally places a greater emphasis on the details of performing testing.

Note that software testing has two core aspects to it:

- *Manual vs. Automated Testing:* In an ideal world, we might love to automate everything, but that's rarely feasible. Some things are simply much better with manual testing because some features are too qualitative for a computer to effectively examine (such as if content represents pornography). Additionally, whereas a computer can generally recognize only issues that it's been told to look for, human observation may reveal new issues that haven't been specifically examined. Both humans and computers form an essential part of the testing process.
- *Black Box Testing vs. White Box Testing:* This distinction refers to the degree of access we have into the software. In black box testing, we're just given the software as-is and need to test it. With white box testing, we have additional programmatic access to test individual functions. We can also automate some black box testing, although it's certainly much harder.

Let's walk through an approach from start to end.

Step 1: Are we doing Black Box Testing or White Box Testing?

Though this question can often be delayed to a later step, I like to get it out of the way early on. Check with your interviewer as to whether you're doing black box testing or white box testing—or both.

Step 2: Who will use it? And why?

Software typically has one or more target users, and the features are designed with this in mind. For example, if you're asked to test software for parental controls on a web browser, your target users include both parents (who are implementing the blocking) and children (who are the recipients of blocking). You may also have "guests" (people who should neither be implementing nor receiving blocking).

Step 3: What are the use cases?

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

Remember that it's not up to you to just magically decide the use cases. This is a conversation to have with your interviewer.

Step 4: What are the bounds of use?

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

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

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

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

Here is where the distinctions between manual and automated testing, and between black box and white box testing, really come into play.

Steps 3 and 4 should have roughly defined the use cases. In step 6, we further define them and discuss how to perform the testing. What exact situations are you testing? Which of these steps can be automated? Which require human intervention?

Remember that while automation allows you to do some very powerful testing, it also has some significant drawbacks. Manual testing should usually be part of your test procedures.

When you go through this list, don't just rattle off every scenario you can think of. It's disorganized, and you're sure to miss major categories. Instead, approach this in a structured manner. Break down your testing into the main components, and go from there. Not only will you give a more complete list of test cases, but you'll also show that you're a structured, methodical person.

Testing a Function

In many ways, testing a function is the easiest type of testing. The conversation is typically briefer and less vague, as the testing is usually limited to validating input and output.

However, don't overlook the value of some conversation with your interviewer. You should discuss any assumptions with your interviewer, particularly with respect to how to handle specific situations.

Suppose you were asked to write code to test `sort(int[] array)`, which sorts an array of integers. You might proceed as follows.

Step 1: Define the test cases

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

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

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

Step 2: Define the expected result

Often, the expected result is obvious: the right output. However, in some cases, you might want to validate additional aspects. For instance, if the `sort` method returns a new sorted copy of the array, you should probably validate that the original array has not been touched.

Step 3: Write test code

Once you have the test cases and results defined, writing the code to implement the test cases should be fairly straightforward. Your code might look something like:

```
void testAddThreeSorted() {
  MyList list = new MyList();
  list.addThreeSorted(3, 1, 2); // Adds 3 items in sorted order
  assertEquals(list.getElement(0), 1);
  assertEquals(list.getElement(1), 2);
  assertEquals(list.getElement(2), 3);
}
```

Troubleshooting Questions

A final type of question is explaining how you would debug or troubleshoot an existing issue. Many candidates balk at a question like this, giving unrealistic answers like "reinstall the software." You can approach these questions in a structured manner, like anything else.

Let's walk through this problem with an example: You're working on the Google Chrome team when you receive a bug report: Chrome crashes on launch. What would you do?

Reinstalling the browser might solve this user's problem, but it wouldn't help the other users who might be experiencing the same issue. Your goal is to understand what's *really* happening, so that the developers can fix it.

Step 1: Understand the Scenario

The first thing you should do is ask questions to understand as much about the situation as possible.

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

Step 2: Break Down the Problem

Now that you understand the details of the scenario, you want to break down the problem into testable units. In this case, you can imagine the flow of the situation as follows:

1. Go to Windows Start menu.
2. Click on Chrome icon.
3. Browser instance starts.
4. Browser loads settings.
5. Browser issues HTTP request for homepage.

6. Browser gets HTTP response.
7. Browser parses webpage.
8. Browser displays content.

At some point in this process, something fails and it causes the browser to crash. A strong tester would iterate through the elements of this scenario to diagnose the problem.

Step 3: Create Specific, Manageable Tests

Each of the above components should have realistic instructions—things that you can ask the user to do, or things that you can do yourself (such as replicating steps on your own machine). In the real world, you will be dealing with customers, and you can't give them instructions that they can't or won't do.

Interview Questions

11.1 **Mistake:** Find the mistake(s) in the following code:

```
unsigned int i;
for (i = 100; i >= 0; --i)
   printf("%d\n", i);
```

Hints: #257, #299, #362

pg 417

11.2 **Random Crashes:** You are given the source to an application which crashes when it is run. After running it ten times in a debugger, you find it never crashes in the same place. The application is single threaded, and uses only the C standard library. What programming errors could be causing this crash? How would you test each one?

Hints: #325

pg 417

11.3 **Chess Test:** We have the following method used in a chess game: `boolean canMoveTo(int x, int y)`. This method is part of the `Piece` class and returns whether or not the piece can move to position `(x, y)`. Explain how you would test this method.

Hints: #329, #401

pg 418

11.4 **No Test Tools:** How would you load test a webpage without using any test tools?

Hints: #313, #345

pg 419

11.5 **Test a Pen:** How would you test a pen?

Hints: #140, #164, #220

pg 420

11.6 **Test an ATM:** How would you test an ATM in a distributed banking system?

Hints: #210, #225, #268, #349, #393

pg 421

Hints start on page 662.

12

C and C++

A good interviewer won't demand that you code in a language you don't profess to know. Hopefully, if you're asked to code in C++, it's listed on your resume. If you don't remember all the APIs, don't worry—most interviewers (though not all) don't care that much. We do recommend, however, studying up on basic C++ syntax so that you can approach these questions with ease.

Classes and Inheritance

Though C++ classes have similar characteristics to those of other languages, we'll review some of the syntax below.

The code below demonstrates the implementation of a basic class with inheritance.

```
#include <iostream>
using namespace std;

#define NAME_SIZE 50 // Defines a macro

class Person {
  int id; // all members are private by default
  char name[NAME_SIZE];

 public:
  void aboutMe() {
    cout << "I am a person.";
  }
};

class Student : public Person {
 public:
  void aboutMe() {
    cout << "I am a student.";
  }
};

int main() {
  Student * p = new Student();
  p->aboutMe(); // prints "I am a student."
  delete p; // Important! Make sure to delete allocated memory.
  return 0;
}
```

All data members and methods are private by default in C++. One can modify this by introducing the keyword `public`.

Constructors and Destructors

The constructor of a class is automatically called upon an object's creation. If no constructor is defined, the compiler automatically generates one called the Default Constructor. Alternatively, we can define our own constructor.

If you just need to initialize primitive types, a simple way to do it is this:

```
Person(int a) {
  id = a;
}
```

This works for primitive types, but you might instead want to do this:

```
Person(int a) : id(a) {
  ...
}
```

The data member `id` is assigned before the actual object is created and before the remainder of the constructor code is called. This approach is necessary when the fields are constant or class types.

The destructor cleans up upon object deletion and is automatically called when an object is destroyed. It cannot take an argument as we don't explicitly call a destructor.

```
~Person() {
  delete obj; // free any memory allocated within class
}
```

Virtual Functions

In an earlier example, we defined p to be of type `Student`:

```
Student * p = new Student();
p->aboutMe();
```

What would happen if we defined p to be a `Person*`, like so?

```
Person * p = new Student();
p->aboutMe();
```

In this case, "`I am a person`" would be printed instead. This is because the function `aboutMe` is resolved at compile-time, in a mechanism known as *static binding*.

If we want to ensure that the `Student`'s implementation of `aboutMe` is called, we can define `aboutMe` in the `Person` class to be `virtual`.

```
class Person {
  ...
  virtual void aboutMe() {
    cout << "I am a person.";
  }
};

class Student : public Person {
 public:
  void aboutMe() {
    cout << "I am a student.";
  }
```