

# HACKING WITH SWIFT



# SWIFT CODING CHALLENGES

**REAL PROBLEMS, REAL SOLUTIONS**

Prepare for iOS interviews,  
test yourself against friends,  
and level up your skills.

Paul Hudson

# Swift Coding Challenges

*Paul Hudson*

# Contents

<b>Introduction</b>	<b>5</b>
Welcome	
<b>Strings</b>	<b>12</b>
Challenge 1: Are the letters unique?	
Challenge 2: Is a string a palindrome?	
Challenge 3: Do two strings contain the same characters?	
Challenge 4: Does one string contain another?	
Challenge 5: Count the characters	
Challenge 6: Remove duplicate letters from a string	
Challenge 7: Condense whitespace	
Challenge 8: String is rotated	
Challenge 9: Find pangrams	
Challenge 10: Vowels and consonants	
Challenge 11: Three different letters	
Challenge 12: Find longest prefix	
Challenge 13: Run-length encoding	
Challenge 14: String permutations	
Challenge 15: Reverse the words in a string	
<b>Numbers</b>	<b>55</b>
Challenge 16: Fizz Buzz	
Challenge 17: Generate a random number in a range	
Challenge 18: Recreate the pow() function	
Challenge 19: Swap two numbers	
Challenge 20: Number is prime	
Challenge 21: Counting binary ones	
Challenge 22: Binary reverse	
Challenge 23: Integer disguised as string	
Challenge 24: Add numbers inside a string	
Challenge 25: Calculate a square root by hand	
Challenge 26: Subtract without subtract	
<b>Files</b>	<b>92</b>
Challenge 27: Print last lines	
Challenge 28: Log a message	

Challenge 29: Documents directory  
Challenge 30: New JPEGs  
Challenge 31: Copy recursively  
Challenge 32: Word frequency  
Challenge 33: Find duplicate filenames  
Challenge 34: Find executables  
Challenge 35: Convert images  
Challenge 36: Print error lines

## Collections

127

Challenge 37: Count the numbers  
Challenge 38: Find N smallest  
Challenge 39: Sort a string array by length  
Challenge 40: Missing numbers in array  
Challenge 41: Find the median  
Challenge 42: Recreate firstIndex(of:)  
Challenge 43: Linked lists  
Challenge 44: Linked list mid-point  
Challenge 45: Traversing the tree  
Challenge 46: Recreate map()  
Challenge 47: Recreate min()  
Challenge 48: Implement a deque data structure  
Challenge 49: Sum the even repeats  
Challenge 50: Count the largest range  
Challenge 51: Reversing linked lists  
Challenge 52: Sum an array of numbers  
Challenge 53: Linked lists with a loop  
Challenge 54: Binary search trees

## Algorithms

195

Challenge 55: Bubble sort  
Challenge 56: Insertion sort  
Challenge 57: Isomorphic values  
Challenge 58: Balanced brackets  
Challenge 59: Quicksort  
Challenge 60: Tic-Tac-Toe winner  
Challenge 61: Find prime numbers  
Challenge 62: Points to angles  
Challenge 63: Flood fill  
Challenge 64: N Queens

## Appendix: Be the Compiler

246

# Introduction

About this book

# Welcome

This is not your average book about Swift coding. This is a book where I expect *you* to do most of the work. Sure, I'll be helping you along with hints, and I'll also be providing my own solutions and explanations along the way, but if you haven't already put the work in then you'll be missing out.

You see, this book is called *Swift Coding Challenges* because I really want to challenge you. There is no learning without struggle, so if you don't take the time to read each challenge and try it for yourself in Xcode, you'll never know how you would have fared.

So, please follow these instructions to the letter if you want to reap the full benefit of what this book offers. It's not a cheatsheet, a guide book, or even a tutorial like I would normally write. Instead, this is a book designed to make you think, to make you work, and to make you *learn by doing*.

Note: this book is not for Swift beginners, so you should have at least six months of Swift behind you. If you've completed Hacking with Swift you ought to be able to handle all the Easy and some of the Tricky problems. If you've completed Pro Swift you ought to be able to handle most of the Taxing problems too. If you consistently struggle with challenges, or if you've never seen the **rethrows** keyword before, you should definitely watch my Pro Swift videos.

Where I thought it was useful, I have used Big O notation to describe the computational complexity of an algorithm. The order of a function (the "O") tells you how fast it grows, and provides a worst-case scenario of how much work must be done to run some code. An O(n) function means "the function takes as correspondingly longer as you add items." So, if it takes one second with one item, it would take 10 seconds with ten items. In comparison, an O(1) function runs in constant time, which means a function that takes 1 second with one item would take 1 second with 100 items.

## How to use this book

I've organized this book into chapters so that it's roughly grouped by different kinds of problem. There is, inevitably, some crossover between chapters, but I've tried to place things where I thought best. Inside each chapter are individual challenges sorted by difficulty, so you ought to be able to flip to a chapter that interests you and just start working your way through from the beginning.

Each challenge is broken down into three parts. First, I state the problem as clearly as possible in one sentence. When necessary I'll provide some extra clarification. When working with strings, I've made it clear whether you should ignore letter case (so "CAT" is equal to "cat") or whether you should take letter case into account (so "CAT" is not equal to "cat").

Second, I provide some example input and output so you should be able to write test cases to validate that your code is correct. I'm not saying that you need to use test-driven development, but I would suggest that if you're going to a job interview TDD is a good skill to be able to show.

After these two steps, I encourage you to fire up Xcode and start coding. How long it takes to solve each challenge depends on your skill level, but I would suggest the following as a rough guide:

1. Novice developers, i.e. under one year of coding experience, Swift or otherwise: 15 minutes for an Easy challenge, 30 minutes for a Tricky challenge, and 1 hour for a Taxing challenge.
2. Intermediate developers, i.e. under three years of coding experience: 10 minutes for an Easy challenge, 20 minutes for a Tricky challenge, and 30 minutes for a Taxing challenge.
3. Senior developers, i.e. five years or more of coding experience: 5 minutes for an Easy challenge, 10 minutes for a Tricky challenge, and 15 minutes for a Taxing challenge.

If you fall somewhere between those groups, I'm sure you're smart enough to extrapolate a rough goal for yourself. If you're way beyond five years of experience then I would expect the times might come down further – perhaps as low as five minutes for a taxing challenge if you're confident.

## Introduction

Obvious warning: the groupings are very broad, so don't worry if you go over the suggested times. I personally would count writing tests and commenting your code in those times, but that's down to you.

As you work, you might find you hit problems – and that's OK. Remember the whole “no learning without struggle” thing? If you hit a brick wall and you're not sure how to continue, every challenge comes with hints from me to try to point you in the right direction. You should read these only if you've tried and failed to complete the solution, and even then read only one hint at a time – you might find just reading the first one is enough to help you advance.

Once you have completed the challenge – which might be when you've written a solution that passes your tests, or might be when your self-allotted time target has lapsed – then it's time for you to read my solution. I've tried to make sure every solution is as clear as possible, so sometimes I've used three lines of code rather than one to allow me to add more comments or discussion. Remember, interview environments are stressful enough without you striving for the perfect answer to a question – get something that works then improve on it, rather than trying to get it into a magic one liner in your first pass!

Many challenges come with more than one solution. This is sometimes to help you compare performance characteristics, sometimes because I can never resist the opportunity to teach new things, but sometimes also because I want to encourage you to be open-minded – there are lots of ways of making things work, and I hope you can appreciate the variety!

## Passing a challenge

I'm afraid there's no certificate when you complete all the challenges, but if you tweet me @twostraws I'll high five you over the internet.

Since I'm not on hand to mark your answers, it's down to you to self-regulate – you're only cheating yourself, after all. I suggest you write tests for your challenges; something like this ought to work, replacing the “X” with your current challenge number:

```
func yourFunctionName(input: String) -> Bool {  
    return true
```

```
}
```

```
assert(yourFunctionName(input: "Hacking with Swift") == true,  
"Challenge X did not return true")
```

I frequently use **challengeX()** for the names of my functions and methods, but only because it makes it easier for you to remember where they came from if you copy them into your own playgrounds.

You should be able to complete most of the challenges in a Swift playground. The Files chapter requires you to work with external files on your computer, so for those challenges you should use a macOS Command Line Tool project.

**Note:** sometimes it's possible you'll fly through a challenge graded as taxing, or struggle with a problem that's graded easy. When this happens, please don't send me angry emails:

1. Things that are easy for you aren't always easy for others, and things that are hard for you aren't always hard for others.
2. The nature of strings make them easier to work with than collections, so you'll find many more easy problems with strings than with collections.
3. What seems easy over a glass of wine on a Sunday evening can seem insurmountable when faced with three interviewers staring at you as you pick up a whiteboard pen!

Often I refer to a solution as being naïve. This is not an insult! If your solution is the same as my naïve solution, it means you totally solved the problem and deserve a pat on the back. It does, however, mean that there are more efficient or cleaner solutions available, and I hope everyone will learn at least a little bit while reading this book.

## A note on algorithms

Some of you reading this won't have had formal computer science education, and perhaps might not have had the chance to fill any gaps on the job. If this is you, it's very likely you might find the algorithms chapter particularly difficult – if you can't tell a bubble sort from an

## Introduction

insertion sort, then it's going to be hard to answer interview questions about them.

Take heart! I do my best to explain algorithms in particular detail, and hope my explanations will help you understand how they work. If you're able to try these challenges that's awesome, but don't feel discouraged if you struggle and end up relying on my solutions particularly heavily. We all have things we don't know – even me.

## Frequent Flyer Club

You can buy Swift tutorials from anywhere, but I'm pleased, proud, and very grateful that you chose mine. I want to say thank you, and the best way I have of doing that is by giving you bonus content above and beyond what you paid for – you deserve it!

Every book contains a word that unlocks bonus content for Frequent Flyer Club members. The word for this book is **COBALT**. Enter that word, along with words from any other Hacking with Swift books, here: <https://www.hackingwithswift.com/frequent-flyer>

## Dedication

This book is dedicated to Taylor Smith, who frequently reads my Taylor Swift-inspired coding examples and imagines they are about him. Well, this time it really *is* about you:

```
import Foundation

let TAYLOR = { (t: Int) in return -(~t) }
let tayloR = { return TAYLOR(TAYLOR($0)) }
let taylorOr = { return tayloR(tayloR($0)) }
let tayLor = { return taylorOr(taylorOr($0)) }
let taYlor = { return tayLor(tayLor($0)) }
let tAylor = { return taYlor(taylorOr($0)) }
let Taylor = { return tAylor(tAylor($0)) }
let taylor = { (t: Int) -> Int in print(UnicodeScalar(t)!, terminator: ""); return 0 }
```

## Copyright

Swift, the Swift logo, Xcode, Instruments, Cocoa Touch, Touch ID, AirDrop, iBeacon, iPhone, iPad, Safari, App Store, Mac and macOS are trademarks of Apple Inc., registered in the U.S. and other countries.

Swift Coding Challenges and Hacking with Swift are copyright Paul Hudson. All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

# **Chapter 1**

## Strings

# Challenge 1: Are the letters unique?

**Difficulty:** Easy

Write a function that accepts a **String** as its only parameter, and returns true if the string has only unique letters, taking letter case into account.

## Sample input and output

- The string “No duplicates” should return true.
- The string “abcdefghijklmnopqrstuvwxyz” should return true.
- The string “AaBbCc” should return true because the challenge is case-sensitive.
- The string “Hello, world” should return false because of the double Ls and double Os.

For this initial challenge I’ll write some test cases for you, so that you have something to use in the future. These four **assert()** statements should all evaluate to true, and therefore not trigger an error:

```
assert(challenge1(input: "No duplicates") == true, "Challenge 1 failed")
assert(challenge1(input: "abcdefghijklmnopqrstuvwxyz") == true, "Challenge 1 failed")
assert(challenge1(input: "AaBbCc") == true, "Challenge 1 failed")
assert(challenge1(input: "Hello, world") == false, "Challenge 1 failed")
```

## Hints

Remember, read as few hints as you can to help you solve the challenge, and only read them if

## Strings

you've tried and failed. (This reminder won't be repeated again.)

**Hint #1:** You should treat the input string like an array that contains **Character** elements.

**Hint #2:** You could use a temporary array to store characters that have been checked, but it's not necessary.

**Hint #3:** Sets are like arrays, except they can't contain duplicate elements.

**Hint #4:** You can create sets from arrays and arrays from sets. Both have a **count** property.

## Solution

There are two ways to solve this, both of which are perfectly fine given our test cases. First, the brute force approach: create an array of checked characters, then loop through every letter in the input string and append the latter to the list of checked characters, returning false as soon as a call to **contains()** fails.

Here's how that code would look:

```
func challenge1a(input: String) -> Bool {
    var usedLetters = [Character]()

    for letter in input {
        if usedLetters.contains(letter) {
            return false
        }

        usedLetters.append(letter)
    }

    return true
}
```

## Challenge 1: Are the letters unique?

That solution is correct with the example input and output I provided, but you should be prepared to discuss that it doesn't scale well: calling `contains()` on an array is an  $O(n)$  operation, which means it gets slower as more items are added to the array. If our text were in a language with very few duplicate characters, such as Chinese, this might cause performance issues.

The smart solution is to use `Set`, which can be created directly from the input string. Sets cannot contain duplicate items, so if we create a set from the input string then the count of the set will equal the count of the input if there are no duplicates.

In code you would write this:

```
func challenge1b(input: String) -> Bool {
    return Set(input).count == input.count
}
```

# Challenge 2: Is a string a palindrome?

**Difficulty:** Easy

Write a function that accepts a **String** as its only parameter, and returns true if the string reads the same when reversed, ignoring case.

## Sample input and output

- The string “rotator” should return true.
- The string “Rats live on no evil star” should return true.
- The string “Never odd or even” should return false; even though the letters are the same in reverse the spaces are in different places.
- The string “Hello, world” should return false because it reads “dlrow ,olleH” backwards.

## Hints

**Hint #1:** You can reverse strings using their **reversed()** method.

**Hint #2:** Two strings compare as equal if they contain the same letters in the same order. They are value types in Swift, so it doesn’t matter how they were created, as long as their values are the same.

**Hint #3:** You need to ignore case, so consider forcing the string to either lowercase or uppercase before comparing.

## Solution

This is one of the most common interview questions you’ll come across, and but fortunately

## Challenge 2: Is a string a palindrome?

it's nice and easy to solve thanks to the **reversed()** method.

One small wrinkle is that **reversed()** doesn't actually return a string for performance reasons, so you need to convert it to a string in order to perform the check, like this:

```
func challenge2a(input: String) -> Bool {
    return String(input.reversed()) == input
}
```

Remember, strings are value types in Swift, which means they compare as equal as long as their contents are identical - it doesn't matter how they are created.

As an analogy, we all know that 2 times 2 is equal to  $2 + 2$ , even though the number 4 was created using different methods. The same is true of Swift's string: even though one is reversed, the `==` operator just compares the current value.

Finally, make sure you remember that your comparison should ignore string case. This can be done with the **lowercased()** method on the input string, like this:

```
func challenge2b(input: String) -> Bool {
    let lowercased = input.lowercased()
    return String(lowercased.reversed()) == lowercased
}
```

Done!

# Challenge 3: Do two strings contain the same characters?

**Difficulty:** Easy

Write a function that accepts two **String** parameters, and returns true if they contain the same characters in any order taking into account letter case.

## Sample input and output

- The strings “abca” and “abca” should return true.
- The strings “abc” and “cba” should return true.
- The strings “ a1 b2 ” and “ b1 a2 ” should return true.
- The strings “abc” and “abca” should return false.
- The strings “abc” and “Abc” should return false.
- The strings “abc” and “cbAa” should return false.
- The strings “abcc” and “abca” should return false.

## Hints

**Hint #1:** This task requires you to handle duplicate characters.

**Hint #2:** The naive way to check this is to loop over the characters in one and check it exists in the other, removing matches as you go.

**Hint #3:** An easier solution is to treat both strings as character arrays.

**Hint #4:** If you sort two character arrays, then you will have something that is the same length and identical character for character.

## Challenge 3: Do two strings contain the same characters?

### Solution

You could write a naïve solution to this problem by taking a variable copy of the second input string, then looping over the first string and checking each letter exists in the second. If it does, remove it so it won't be counted again; if not, return false. If you get to the end of the first string, then return true if the second string copy is now empty, otherwise return false.

For example:

```
func challenge3a(string1: String, string2: String) -> Bool {
    var checkString = string2

    for letter in string1 {
        if let index = checkString.firstIndex(of: letter) {
            checkString.remove(at: index)
        } else {
            return false
        }
    }

    return checkString.count == 0
}
```

That solution works, but is less than ideal because you're having to look up letter positions repeatedly using **firstIndex(of:)**, which is O(n). Worse, the **remove(at:)** call is also O(n), because it needs to move other elements down in the array once the item is removed.

A simpler solution is to remember that strings are sequences in Swift and sort them directly. Once that's done, you can do a direct comparison using `==`. This ends up involving much less code:

```
func challenge3b(string1: String, string2: String) -> Bool {
    return string1.sorted() == string2.sorted()
}
```

## Strings

# Challenge 4: Does one string contain another?

**Difficulty:** Easy

Write your own version of the **contains()** method on **String** that ignores letter case, and without using the existing **contains()** method.

## Sample input and output

- The code "Hello, world".fuzzyContains("Hello") should return true.
- The code "Hello, world".fuzzyContains("WORLD") should return true.
- The code "Hello, world".fuzzyContains("Goodbye") should return false.

## Hints

**Hint #1:** You should write this as an extension to **String**.

**Hint #2:** You can't use **contains()**, but there are other methods that do similar things.

**Hint #3:** Try the **range(of:)** method.

**Hint #4:** To ignore case, you can either uppercase both strings, or try the second parameter to **range(of:)**.

## Solution

If you were already familiar with the **range(of:)** method, this one should have proved straightforward. If not, you were probably wondering why I gave it an easy grade!

The **range(of:)** method returns the position of one string inside another. As it's possible the

## Strings

substring might not exist in the other, the return value is optional. This is perfect for us: if we call **range(of:)** and get back nil, it means the substring isn't contained inside the check string.

Ignoring letter case adds a little complexity, but can be solved either by collapsing the case before you do your check, or by using the **.caseInsensitive** option for **range(of:)**.

The former looks like this:

```
extension String {  
    func fuzzyContains(_ string: String) -> Bool {  
        return self.uppercased().range(of: string.uppercased()) !=  
            nil  
    }  
}
```

And the latter like this:

```
extension String {  
    func fuzzyContains(_ string: String) -> Bool {  
        return range(of: string, options: .caseInsensitive) !=  
            nil  
    }  
}
```

In this instance the two are identical, but there's a benefit to collapsing the case if you had to check through lots of items.

# Challenge 5: Count the characters

**Difficulty:** Easy

Write a function that accepts a string, and returns how many times a specific character appears, taking case into account.

**Tip:** If you can solve this without using a **for-in** loop, you can consider it a Tricky challenge.

## Sample input and output

- The letter “a” appears twice in “The rain in Spain”.
- The letter “i” appears four times in “Mississippi”.
- The letter “i” appears three times in “Hacking with Swift”.

## Hints

**Hint #1:** Remember that **String** and **Character** are different data types.

**Hint #2:** Don’t be afraid to go down the brute force route: looping over characters using a **for-in** loop.

**Hint #3:** You could solve this functionally using **reduce()**, but tread carefully.

**Hint #4:** You could solve this using **NSCountedSet**, but I’d be suspicious unless you could justify the extra overhead.

## Solution

You might be surprised to hear me saying this, but: this is a great interview question. It’s simple to explain, it’s simple to code, and it has enough possible solutions that it’s likely to generate some interesting discussion – which is gold dust in interviews.

## Strings

This question is also interesting, because it's another good example where the simple brute force approach is both among the most readable and most efficient. I suggested two alternatives in the hints, and I think it's an interesting code challenge for you to try all three.

First, the easy solution: loop over the characters by hand, comparing against the check character. In code, it would be this:

```
func challenge5a(input: String, count: Character) -> Int {  
    var letterCount = 0  
  
    for letter in input {  
        if letter == count {  
            letterCount += 1  
        }  
    }  
  
    return letterCount  
}
```

There's nothing complicated there, but do make sure you accept the check character as a **Character** to make the equality operation smooth.

The second option is to solve this problem functionally using **reduce()**. This has the advantage of making for very clear, expressive, and concise code, particularly when combined with the ternary operator:

```
func challenge5b(input: String, count: Character) -> Int {  
    return input.reduce(0) {  
        $1 == count ? $0 + 1 : $0  
    }  
}
```

So, that will start with 0, then go over every character in the string. If a given letter matches the

## Challenge 5: Count the characters

input character, then it will add 1 to the reduce counter, otherwise it will return the current reduce counter. Functional programming does make for shorter code, and the intent here is nice and clear, however this is not quite as performant – it's likely to run about 10% slower than the first solution depending on your configuration.

A third solution is to use **NSCountedSet**, but that's wasteful unless you intend to count several characters. It's also complicated because Swift bridges **String** to **NSObject** well, but doesn't bring **Character**, so **NSCountedSet** won't play nicely unless you convert the characters yourself. So, your code would end up being something like this:

```
func challenge5c(input: String, count: String) -> Int {
    let array = input.map { String($0) }
    let counted = NSCountedSet(array: array)

    return counted.count(for: count)
}
```

That creates an array of strings by converting each character in the input string, then creates a counted set from the string array, and finally returns the count – for a single letter. Wasteful, for sure, and inefficient too – a massive ten times slower than the original.

There's actually a fourth option you might have chosen. It's the shortest option, however it requires a little lateral thinking: you can calculate how many times a letter appears in a string by removing it, then comparing the lengths of the original and modified strings. Here it is in Swift:

```
func challenge5d(input: String, count: String) -> Int {
    let modified = input.replacingOccurrences(of: count, with:
    ""))
    return input.count - modified.count
}
```

# Challenge 6: Remove duplicate letters from a string

**Difficulty:** Easy

Write a function that accepts a string as its input, and returns the same string just with duplicate letters removed.

**Tip:** If you can solve this challenge without a **for-in** loop, you can consider it “tricky” rather than “easy”.

## Sample input and output

- The string “wombat” should print “wombat”.
- The string “hello” should print “helo”.
- The string “Mississippi” should print “Misp”.

## Hints

**Hint #1:** Sets are great at removing duplicates, but bad at retaining order.

**Hint #2:** Foundation does have a way of forcing sets to retain their order, but you need to handle the typecasting.

**Hint #3:** You can create strings out of character arrays.

**Hint #4:** You can solve this functionally using **filter()**.

## Solution

There are three interesting ways this can be solved, and I’m going to present you with all three

## Challenge 6: Remove duplicate letters from a string

so you can see which suits you best. Remember: “fastest” isn’t always “best”, not least because readability is important, but also particularly because “memorizability” is important too – the perfect solution is often easily forgotten when you’re being tested.

Let’s look at a simple but interesting solution first: using sets. Swift’s standard library has a built-in **Set** type, but it does *not* preserve the order of its elements. This is a shame, because otherwise the solution would have been as simple as this:

```
let string = "wombat"
let set = Set(string)
print(String(set))
```

However, Foundation has a specialized set type called **NSOrderedSet**. This also removes duplicates, but now ensures items stay in the order they were added. Sadly, it’s not bridged to Swift in any pleasing way, which means to use it you must add typecasting: once from **Character** to **String** before creating the set, then once from **Array<Any>** to **Array<String>**.

This function does just that:

```
func challenge6a(string: String) -> String {
    let array = string.map { String($0) }
    let set = NSOrderedSet(array: array)
    let letters = Array(set) as! Array<String>
    return letters.joined()
}
```

That passes all tests, but I think you’ll agree it’s a bit ugly. I suspect Swift might see a native **OrderedSet** type in the future.

A second solution is to take a brute-force approach: create an array of used characters, then loop through every letter in the string and check if it’s already in the used array. If it isn’t, add it, then finally return a stringified form of the used array.

This is nice and easy to write, as long as you know that you can create a **String** directly from a

## Strings

**Character** array:

```
func challenge6b(string: String) -> String {
    var used = [Character]()
    for letter in string {
        if !used.contains(letter) {
            used.append(letter)
        }
    }
    return String(used)
}
```

There is a third solution, and I think it's guaranteed to generate some interesting discussion in an interview or book group!

As you know, dictionaries hold a value attached to a key, and only one value can be attached to a specific key at any time. You can change the value attached to a key just by assigning it again, but you can also call the **updateValue()** method – it does the same thing, but also returns either the original value or nil if there wasn't one. So, if you call **updateValue()** and get back nil it means “that wasn't already in the dictionary, but it is now.”

We can use this method in combination with the **filter()** method on our input string's **character** property: filter the characters so that only those that return nil for **updateValue()** are used in the return array.

So, the third solution to this challenge looks like this:

```
func challenge6c(string: String) -> String {
    var used = [Character: Bool]()
    let result = string.filter {
        used.updateValue(true, forKey: $0) == nil
    }
    return String(result)
}
```

## Challenge 6: Remove duplicate letters from a string

```
    }  
  
    return String(result)  
}
```

As long as you know about the **updateValue()** method, that code is brilliantly readable – the use of **filter()** means it's clear what the loop is trying to do. However, if you *don't* know about **updateValue()** then I suspect it falls short and is best avoided.

# Challenge 7: Condense whitespace

**Difficulty:** Easy

Write a function that returns a string with any consecutive spaces replaced with a single space.

## Sample input and output

I've marked spaces using "[space]" below for visual purposes:

- The string "a[space][space][space]b[space][space][space]c" should return "a[space]b[space]c".
- The string "[space][space][space][space]a" should return "[space]a".
- The string "abc" should return "abc".

## Hints

**Hint #1:** You might think it a good idea to use **components(separatedBy:)** then **joined()**, but that will struggle with leading and trailing spaces.

**Hint #2:** You could loop over each character, keeping track of a **seenSpace** boolean that gets set to true when the previous character was a space.

**Hint #3:** You could use regular expressions.

**Hint #4:** Try using **replacingOccurrences(of:)**

## Solution

As is the case for many other string challenges, we can write a naïve solution or a clever one,

## Challenge 7: Condense whitespace

but here the clever one is dramatically simpler – and it uses regular expressions. (Yes, you *did* just read “simpler” and “regular expressions” in the same sentence.)

But first, let’s look at something you might have tried:

```
func challenge7(input: String) -> String {
    let components =
input.components(separatedBy: .whitespacesAndNewlines)
    return components.filter { !$0.isEmpty }.joined(separator: " ")
}
```

That splits a string up by its spaces, then removes any empty items, and joins the remainder using a space, and is the ideal solution – if your goal is to remove any duplicate whitespace while *also* removing leading and trailing whitespace. However, it fails the requirement that “[space][space][space][space]a” should return “[space]a“, so you should have rejected it.

Instead, you might have written a loop over the characters in the input string. If the current letter was a space and you had already seen one in this run, continue to the next letter. Otherwise, mark that you’ve seen a space. If it wasn’t a space, clear the space flag. Regardless of whether it was the first space or a letter, append it to an output string.

Transform that into Swift and you get this:

```
func challenge7a(input: String) -> String {
    var seenSpace = false
    var returnValue = ""

    for letter in input {
        if letter == " " {
            if seenSpace { continue }
            seenSpace = true
        } else {
            seenSpace = false
            returnValue += letter
        }
    }
    return returnValue
}
```

## Strings

```
    }

    returnValue.append(letter)
}

return returnValue
}
```

This is a clear solution, and it works great. However, for once, this is a place where regular expressions can help: they turn all that into a single line of code:

```
func challenge7b(input: String) -> String {
    return input.replacingOccurrences(of: " +", with: " ",
options: .regularExpression, range: nil)
}
```

If you're not familiar with regular expressions, "[space]+" means "match one or more spaces", so that will cause all multiple spaces to be replaced with a single space. Running regular expressions isn't cheap, so that code runs about 50% the speed of the manual solution, but you would have to be doing a heck of a lot of work in order for it to be noticeable.

# Challenge 8: String is rotated

**Difficulty:** Tricky

Write a function that accepts two strings, and returns true if one string is rotation of the other, taking letter case into account.

**Tip:** A string rotation is when you take a string, remove some letters from its end, then append them to the front. For example, “swift” rotated by two characters would be “ftswi”.

## Sample input and output

- The string “abcde” and “eabcd” should return true.
- The string “abcde” and “cdeab” should return true.
- The string “abcde” and “abced” should return false; this is not a string rotation.
- The string “abc” and “a” should return false; this is not a string rotation.

## Hints

**Hint #1:** This is easier than you think.

**Hint #2:** A string is only considered a rotation if it is identical to the original once you factor in the letter movement. That is, “tswi” is not a rotation of “swift” because it is missing the F.

**Hint #3:** If you write a string twice, it must encapsulate all possible rotations, e.g. “catcat” contains “cat”, “tca”, and “atc”.

## Solution

This question appears in coding interviews far more than it deserves, because it’s a problem that seems tricky the first time you face it but is staring-you-in-the-face obvious once someone has told you the solution. I wonder how many times this question appears on interviews just so

## Strings

the interviewer can feel smug about knowing the answer!

Anyway, let's talk about the solution. As I said in hint #3, if you write a string twice it must always encapsulate all possible rotations. So if your string was "abc" then you would double it to "abcabc", which contains all possible rotations: "abc", "cab", and "bca".

So, an initial solution might look like this:

```
func challenge8(input: String, rotated: String) -> Bool {  
    let combined = input + input  
    return combined.contains(rotated)  
}
```

However, that's imperfect – the final example input and output was that "abc" should return false when given the test string "a". Using the code above, the input string would be doubled to "abcabc", which clearly contains the test string "a". To fix this, we need to check not only that the test string exists in the doubled input, but also that both strings are the same size.

So, the correct solution is this:

```
func challenge8(input: String, rotated: String) -> Bool {  
    guard input.count == rotated.count else { return false }  
    let combined = input + input  
    return combined.contains(rotated)  
}
```

Like I said, it's easier than you think, but is it a test of coding knowledge? Not really. If anything, you get a brief "aha!" flash when someone explains the solution to you, but apart from scoring you some interview brownie points I doubt this would be useful in real life.

# Challenge 9: Find pangrams

**Difficulty:** Tricky

Write a function that returns true if it is given a string that is an English pangram, ignoring letter case.

**Tip:** A pangram is a string that contains every letter of the alphabet at least once.

## Sample input and output

- The string “The quick brown fox jumps over the lazy dog” should return true.
- The string “The quick brown fox jumped over the lazy dog” should return false, because it’s missing the S.

## Hints

**Hint #1:** Make sure you start by collapsing case using something like `lowercased()`.

**Hint #2:** You can compare letters using `>`, `>=`, and so on.

**Hint #3:** If you remove duplicates and non-alphabetic characters, the remaining string should add up to 26 letters.

## Solution

You could try and solve this using character sets, but it’s really not needed: Swift’s characters conform to **Comparable**, so you can compare them against “a” and “z” directly to ensure they are alphabetical.

Once you know how to ensure a letter is alphabetical, all that remains is removing duplicates (easy using a set) and collapsing case (`lowercased()` is fine), then comparing the count of the

## Strings

result against 26.

So, here's an example solution in just three lines of code:

```
func challenge9(input: String) -> Bool {  
    let set = Set(input.lowercased())  
    let letters = set.filter { $0 >= "a" && $0 <= "z" }  
    return letters.count == 26  
}
```

# Challenge 10: Vowels and consonants

**Difficulty:** Tricky

Given a string in English, return a tuple containing the number of vowels and consonants.

**Tip:** Vowels are the letters, A, E, I, O, and U; consonants are the letters B, C, D, F, G, H, J, K, L, M, N, P, Q, R, S, T, V, W, X, Y, Z.

## Sample input and output

- The input “Swift Coding Challenges” should return 6 vowels and 15 consonants.
- The input “Mississippi” should return 4 vowels and 7 consonants.

## Hints

**Hint #1:** Just because a letter is not a vowel, it doesn’t mean it’s a consonant – think punctuation, for example.

**Hint #2:** You’ll need to differentiate carefully between the **String** and **Character** types.

**Hint #3:** You could use **CharacterSet** here, but is it really needed?

**Hint #4:** Your return type should be **(vowels: Int, consonants: Int)**.

**Hint #5:** Watch out for uppercase and lowercase letters – an “A” is a vowel regardless of its case.

## Solution

There are three interesting ways to solve this challenge, and I’m going to present them to you

## Strings

slowest first – although realistically you’ll only see significant performance differences if these functions are being called run a hundred times or more.

First, you could use **CharacterSet** here, but it has a fatal flaw: even though “Character” is right there in the name, you can’t ask a character set whether it contains a single character because it works on a different type called **UnicodeScalar**. Instead, you need to convert the **Character** to a **String**, then use its **rangeOfCharacter(from:)** method, like this:

```
func challenge10a(input: String) -> (vowels: Int, consonants: Int) {
    let vowels = CharacterSet(charactersIn: "aeiou")
    let consonants = CharacterSet(charactersIn:
"bcdfghjklmnpqrstvwxyz")

    var vowelCount = 0
    var consonantCount = 0

    for letter in input.lowercased() {
        let stringLetter = String(letter)

        if stringLetter.rangeOfCharacter(from: vowels) != nil {
            vowelCount += 1
        } else if stringLetter.rangeOfCharacter(from:
consonants) != nil {
            consonantCount += 1
        }
    }

    return (vowelCount, consonantCount)
}
```

The second solution builds on the first: if you’re going to work with string methods, why not just skip the character set entirely? There’s a perfectly good **contains()** method on strings that

## Challenge 10: Vowels and consonants

works just as well, so you can avoid allocating a character set at all:

```
func challenge10b(input: String) -> (vowels: Int, consonants: Int) {
    let vowels = "aeiou"
    let consonants = "bcdfghjklmnpqrstvwxyz"

    var vowelCount = 0
    var consonantCount = 0

    for letter in input.lowercased() {
        let stringLetter = String(letter)

        if vowels.contains(stringLetter) {
            vowelCount += 1
        } else if consonants.contains(stringLetter) {
            consonantCount += 1
        }
    }

    return (vowelCount, consonantCount)
}
```

The third option is to do away with strings entirely: create arrays of vowel characters and consonant characters, then use the `contains()` method on the arrays to see if each letter matches. In code, you'd get this:

```
func challenge10c(input: String) -> (vowels: Int, consonants: Int) {
    let vowels = "aeiou"
    let consonants = "bcdfghjklmnpqrstvwxyz"

    var vowelCount = 0
```

## Strings

```
var consonantCount = 0

for letter in input.lowercased() {
    if vowels.contains(letter) {
        vowelCount += 1
    } else if consonants.contains(letter) {
        consonantCount += 1
    }
}

return (vowelCount, consonantCount)
}
```

There's very little to separate each of these three answers in terms of performance, so go with whichever is most familiar to you.

# Challenge 11: Three different letters

**Difficulty:** Tricky

Write a function that accepts two strings, and returns true if they are identical in length but have no more than three different letters, taking case and string order into account.

## Sample input and output

- The strings “Clamp” and “Cramp” would return true, because there is one letter difference.
- The strings “Clamp” and “Crams” would return true, because there are two letter differences.
- The strings “Clamp” and “Grams” would return true, because there are three letter differences.
- The strings “Clamp” and “Grans” would return false, because there are four letter differences.
- The strings “Clamp” and “Clam” would return false, because they are different lengths.
- The strings “clamp” and “maple” should return false. Although they differ by only one letter, the letters that match are in different positions.

## Hints

**Hint #1:** If you value your sanity, get both strings into arrays as quickly as possible.

**Hint #2:** You probably want to use the `enumerated()` method on one array, to get the index and character at the same time.

**Hint #3:** Your function should return false as soon as it reaches four differences; there’s no point continuing to check characters.

## Strings

**Hint #4:** Make sure you check the strings are the same size first, preferably using **guard**.

## Solution

This problem isn't hard as long as you convert your strings into character arrays – if you don't, you need to advance through string indices, which is never pleasant and certainly hard to do during an interview.

The simplest, clearest way to solve this challenge is like so:

1. Start with an early return in case the two strings have different lengths.
2. Create arrays out of both strings.
3. Initialize a **differences** counter to 0.
4. Loop over the first array, using **enumerated()** so we get the current index as well as each character.
5. Compare that character against the character at the same index in the other string array.
6. If they are different, add one to **differences**.
7. If as a result of that **differences** is now 4, return false.
8. On the other hand, if we get to the end of the array, it means we can return true.

Something like this ought to do the trick:

```
func challenge11(first: String, second: String) -> Bool {  
    guard first.count == second.count else { return false }  
  
    let firstArray = Array(first)  
    let secondArray = Array(second)  
    var differences = 0  
  
    for (index, letter) in firstArray.enumerated() {  
        if secondArray[index] != letter {  
            differences += 1  
        }  
    }  
    return differences <= 3  
}
```

## Challenge 11: Three different letters

```
if differences == 4 {  
    return false  
}  
}  
}  
  
return true  
}
```

# Challenge 12: Find longest prefix

**Difficulty:** Tricky

Write a function that accepts a string of words with a similar prefix, separated by spaces, and returns the longest substring that prefixes all words.

## Sample input and output

- The string “swift switch swill swim” should return “swi”.
- The string “flip flap flop” should return “fl”.

## Hints

**Hint #1:** Start with **components(separatedBy:)** so you can check words with a loop.

**Hint #2:** You’ll need a property for the prefix you’re currently checking as well as for the best prefix you’ve found so far.

**Hint #3:** Make sure you use the **hasPrefix()** method.

## Solution

I’ve watched some people blast through this code in a minute, and others struggle to finish in 30 minutes as they get into a mess of recursion.

The key to a simple solution is the **hasPrefix()** method, which avoids the mess of string slicing: start with an empty string, then continue adding more letters from the first word until **hasPrefix()** fails for any of the other words.

So: rather than trying to write a recursive function, you can solve this problem using an inner loop, like this:

## Challenge 12: Find longest prefix

```
func challenge12(input: String) -> String {
    let parts = input.components(separatedBy: " ")
    guard let first = parts.first else { return "" }

    var currentPrefix = ""
    var bestPrefix = ""

    for letter in first {
        currentPrefix.append(letter)

        for word in parts {
            if !word.hasPrefix(currentPrefix) {
                return bestPrefix
            }
        }
    }

    bestPrefix = currentPrefix
}

return bestPrefix
}
```

# Challenge 13: Run-length encoding

**Difficulty:** Taxing

Write a function that accepts a string as input, then returns how often each letter is repeated in a single run, taking case into account.

**Tip:** This approach is used in a simple lossless compression technique called run-length encoding.

## Sample input and output

- The string “aabbc” should return “a2b2c2”.
- The strings “aaabaaaabaaa” should return “a3b1a3b1a3”
- The string “aaAAaa” should return “a2A2a2”

## Hints

**Hint #1:** This would be quite straightforward in other languages using character lookahead, but that’s expensive in Swift thanks to grapheme clusters – **String** *isn’t* an array.

**Hint #2:** To use the “Swifty” approach of looping over your string as-is, make sure you keep track of the current character, and its count, as you loop over the string.

**Hint #3:** Alternatively, consider converting your string to a proper array that you can index into freely, so you can look ahead to compare letters.

## Solution

There are two ways you could solve this, but in a stressful interview environment realistically

## Challenge 13: Run-length encoding

there's only one you'll reach for: the “dumb”, brute force approach. I put dumb in quotes for a reason – more on that later.

This approach would use an algorithm like this:

- Create a **currentLetter** variable that contains an optional **Character**, as well as a counter integer and a return value string.
- Loop through every letter in the input string.
- If the letter is equal to our current letter, add one to the counter.
- Otherwise, if **currentLetter** has a value it means we already had a letter and it's about to change, so append it to the return string.
- Regardless, update **currentLetter** to be the new letter, and reset the counter to 1.
- Once the loop finishes, append **currentLetter** to the return string along with the counter, then return it.

That last step is easily forgotten – because the return string is only modified when the letter changes, the last letter sequence won't be added unless we do it by hand.

Putting that into code:

```
func challenge13a(input: String) -> String {  
    var currentLetter: Character?  
    var returnValue = ""  
    var letterCounter = 0  
  
    for letter in input {  
        if letter == currentLetter {  
            letterCounter += 1  
        } else {  
            if let current = currentLetter {  
                returnValue.append("\\" + (current) + (letterCounter))  
            }  
        }  
    }  
    return returnValue  
}
```

## Strings

```
        currentLetter = letter
        letterCounter = 1
    }
}

if let current = currentLetter {
    returnValue.append("\(current)\\(letterCounter)")
}

return returnValue
}
```

An alternative solution, which is probably one you would be more likely to use if you had come to Swift after learning a different language, would be to use character look ahead: if the next character is different to the current one, or if we're about to hit the end of the array, then modify the return value and reset the counter.

This alternative solution doesn't come naturally to Swift developers, because indexing into strings is expensive in Swift thanks to the grapheme cluster system. However, you could convert the Swift character index into an array, then use *that*. This would give you the following solution:

```
func challenge13b(input: String) -> String {
    var returnValue = ""
    var letterCounter = 0
    var letterArray = Array(input)

    for i in 0 ..< letterArray.count {
        letterCounter += 1

        if i + 1 == letterArray.count || letterArray[i] != letterArray[i + 1] {
            returnValue += "\(letterArray[i])\\(letterCounter)"
        }
    }
}
```

## Challenge 13: Run-length encoding

```
    letterCounter = 0
}
}

return returnValue
}
```

That's a valid solution, but does it beat the "dumb" one? Well, it depends what you mean by "beat": it's certainly less code, it avoids the **if let** repetition, and it should run a little faster – but could you remember it in an interview situation? Writing fast code is important when you're building for production, but much less important when in an interview!

# Challenge 14: String permutations

**Difficulty:** Taxing

Write a function that prints all possible permutations of a given input string.

**Tip:** A string permutation is any given rearrangement of its letters, for example “boamtw” is a permutation of “wombat”.

## Sample input and output

- The string “a” should print “a”.
- The string “ab” should “ab”, “ba”.
- The string “abc” should print “abc”, “acb”, “bac”, “bca”, “cab”, “cba”.
- The string “wombat” should print 720 permutations.

## Hints

**Hint #1:** Your function will need to call itself.

**Hint #2:** The number of lines printed should be the factorial of the length of your string, e.g. “wombat” has six characters, so will have  $6!$  permutations:  $6 \times 5 \times 4 \times 3 \times 2 \times 1$ , or 720.

**Hint #3:** You’ll find it easiest to convert the string to a character array for easier indexing.

**Hint #4:** Each time your function is called, it should loop through all letters in the string so that all combinations are generated.

**Hint #5:** You can slice arrays using `strArray[0...3]`.

**Hint #6:** You can convert string array slices into strings just by using an initializer:

`String(strArray[0...3])`.

## Solution

This is a recursive function with lots of looping, but the nature of factorials is that the loops get smaller each time. Given the input string “wombat”, the first time the function is called you’ll need to loop from 0 up to 5, calling the recursive function each time.

So, initially you’ll pick “w” as it’s the first letter, and in you go to the recursive function – but this time there are only five letters to loop over, so you choose “o”, and go into the function again, etc.

Eventually you spell out “wombat”, which is the result of choosing the first remaining letter each time. But now that you’ve reached the deepest point of the recursion, you back up a level: when you had “womb” it chose the first letter in the remainder (“at”) to make “wombat”, but now that path has been explored it should choose the second remaining letter (“t”) to make “wombt”, at which point the only remaining letter now is “a” to make “wombta”.

Again the recursion has maxed out, so now it will need to go back one level further: when it was at “wom” the remainder was “bat” so it chose the first letter, but now it should choose the second, to make “woma”, with “bt” as remainder. On the first pass it will choose “b” first then “t” (making “womabt”), and on the second it will choose “t” then “b” (making “womatb”). That subset of the path is now maxed out, so it will wind back to “wom” and “bat” and choose the *third* letter, to make “womt” with “ba” as remainder. So it will get “womtba” then “womtab”, and so on, and so on.

That’s the algorithm. It sounds clunky when explained step by step, but trust me: a CPU *flies* through this. Here it is in code:

```
func challenge14(string: String, current: String = "") {
    let length = string.count
    let strArray = Array(string)

    if (length == 0) {
        // there's nothing left to re-arrange; print the result
    } else {
        for i in 0..

```

## Strings

```
print(current)
print("*****")
} else {
    print(current)

    // loop through every character
    for i in 0 ..< length {
        // get the letters before me
        let left = String(strArray[0 ..< i])

        // get the letters after me
        let right = String(strArray[i+1 ..< length])

        // put those two together and carry on
        challenge14(string: left + right, current: current +
String(strArray[i]))
    }
}
}
```

Note: the `print("*****")` and second `print(current)` call are there to help you see how the function works; they serve no functional purpose.

# Challenge 15: Reverse the words in a string

**Difficulty:** Taxing

Write a function that returns a string with each of its words reversed but in the original order, without using a loop.

## Sample input and output

- The string “Swift Coding Challenges” should return “tfiwS gnidoC segnellahC”.
- The string “The quick brown fox” should return “ehT kciuq nworb xof”.

## Hints

**Hint #1:** You should start by converting the string into an array by separating on spaces.

**Hint #2:** You can reverse the characters in a string by calling **reversed()** on it.

**Hint #3:** You can create a string from a character array.

**Hint #4:** You want to convert an array of left to right strings into an array of right to left strings, all without using a loop - this is a perfect use for **map()**.

**Hint #5:** Once you have an array of reversed strings, you can create a single string using **joined()**.

## Solution

The requirement not to use a loop is what makes this a taxing challenge; if you could just use **for-in** then it would be easy or tricky depending on who spoke to.

## Strings

The only sensible way to solve this is using functional programming: the need to convert from one kind of array (“words written left to right”) to another (“words written right to left”) is the perfect use case for **map()**, so most of what remains is creating the array by splitting on spaces then recreating the string once you’ve reversed the words.

Here’s my solution:

```
func challenge15(input: String) -> String {  
    let parts = input.components(separatedBy: " ")  
    let reversed = parts.map { String($0.reversed()) }  
    return reversed.joined(separator: " ")  
}
```

# **Chapter 2**

## Numbers

# Challenge 16: Fizz Buzz

**Difficulty:** Easy

Write a function that counts from 1 through 100, and prints “Fizz” if the counter is evenly divisible by 3, “Buzz” if it’s evenly divisible by 5, “Fizz Buzz” if it’s even divisible by three *and* five, or the counter number for all other cases.

## Sample input and output

- 1 should print “1”
- 2 should print “2”
- 3 should print “Fizz”
- 4 should print “4”
- 5 should print “Buzz”
- 15 should print “Fizz Buzz”

## Hints

**Hint #1:** You’ll need to use modulus: `%`.

**Hint #2:** Check for the “Fizz Buzz” case first, because that’s most specific.

**Hint #3:** Remember to use the closed range operator to include the number 100 at the end.

## Solution

This is the holotype of interview questions, so it was inevitable it would be in here somewhere. Sadly, people really do fail it – I’ve seen it myself – which ought to be impossible, so I hope you didn’t just skip by blithely!

A simple solution looks like this:

## Challenge 16: Fizz Buzz

```
func challenge16a() {
    for i in 1...100 {
        if i % 3 == 0 && i % 5 == 0 {
            print("Fizz Buzz")
        } else if i % 3 == 0 {
            print("Fizz")
        } else if i % 5 == 0 {
            print("Buzz")
        } else {
            print(i)
        }
    }
}
```

You could make it slightly more efficient by nesting the “Fizz Buzz” case inside two **if** statements rather than one:

```
func challenge16b() {
    for i in 1...100 {
        if i % 3 == 0 {
            if i % 5 == 0 {
                print("Fizz Buzz")
            } else {
                print("Fizz")
            }
        } else if i % 5 == 0 {
            print("Buzz")
        } else {
            print(i)
        }
    }
}
```

## Numbers

Using this approach, you don't end up evaluating `i % 3` twice.

You could be a bit more fancy using `forEach` and ternary operators, but you're basically sacrificing readability for smugness:

```
func challenge16c() {
    (1...100).forEach {
        print($0 % 3 == 0 ? $0 % 5 == 0 ? "Fizz Buzz" : "Fizz" :
$0 % 5 == 0 ? "Buzz" : "\($0)")
    }
}
```

Note that I used "`\($0)`" rather than `String($0)` – this is a pet hate of mine, but that code is so complex that Swift actually fails to compile if I used `String($0)`!

# Challenge 17: Generate a random number in a range

**Difficulty:** Easy

Write a function that accepts positive minimum and maximum integers, and returns a random number between those two bounds, inclusive.

## Sample input and output

- Given minimum 1 and maximum 5, the return values 1, 2, 3, 4, 5 are valid.
- Given minimum 8 and maximum 10, the return values 8, 9, 10 are valid.
- Given minimum 12 and maximum 12, the return value 12 is valid.
- Given minimum 12 and maximum 18, the return value 7 is invalid.

## Hints

**Hint #1:** There are lots of ways to generate random numbers in Swift; you'll be judged – silently or openly – on your choice.

**Hint #2:** This question is really a test of how thoroughly you know Swift. There's a really easy and built-in way to solve this, *if* you know about it.

**Hint #3:** Keep in mind that lots of random number generators generate from 0 up to a certain point so you'll need to write code to count from an arbitrary number upwards.

**Hint #4:** Also remember that lots of random number generators generate up to but *excluding* the maximum, so you should add 1 to make sure your tests pass.

**Hint #5:** Take a look at `arc4random_uniform()`.

## Solution

There are several choices for random number generators, and your choice says a lot about your skill level. Very roughly:

- If you used `rand()` you probably came from a C background, or don't generally care about randomness.
- If you used GameplayKit, you're either fairly new to iOS development, or particularly interested in the random number generation shaping GameplayKit offers.
- If you used `arc4random()` you're showing some awareness of half-decent random number generator, but are unaware of – or uninterested in – modulo bias.
- If you used `arc4random_uniform()`, then you're showing some serious chops.
- If you used Swift's `Int.random(in:)` method then you've found the smartest and simplest solution. I wish more people knew about this method!

Of the five options, `Int.random(in:)` will always be preferable because it's the natural way of solving the problem:

```
func challenge17a(min: Int, max: Int) -> Int {  
    return Int.random(in: min...max)  
}
```

If you *didn't* choose that, `arc4random_uniform()` is preferred amongst developers because it generates suitably random numbers for most purposes, it doesn't require seeding, it isn't prone to modulo bias, and it isn't restricted to Apple platforms – it's a commonly used C function that is well understood.

Using `arc4random_uniform()` has three hiccups in Swift: it generates numbers from 0 up to an upper bound, it *excludes* the upper bound rather than *including* it, and it uses `UInt32` rather than `Int`, so you need some typecasting.

We can fix all those problems and still write just one line of code to solve the challenge:

## Challenge 17: Generate a random number in a range

```
func challenge17b(min: Int, max: Int) -> Int {  
    return Int(arc4random_uniform(UINT32(max - min + 1))) + min  
}
```

# Challenge 18: Recreate the pow() function

**Difficulty:** Easy

Create a function that accepts positive two integers, and raises the first to the power of the second.

**Tip:** If you name your function **myPow()** or **challenge18()**, you'll be able to use the built-in **pow()** for your tests. The built-in **pow()** uses doubles, so you'll need to typecast.

## Sample input and output

- The inputs 4 and 3 should return 64, i.e. 4 multiplied by itself 3 times.
- The inputs 2 and 8 should return 256, i.e. 2 multiplied by itself 8 times.

## Hints

**Hint #1:** You don't need any hints to solve this one.

**Hint #2:** Oh, alright: here's a hint: you can either use a loop or, if you're feeling fancy, use a recursive function.

**Hint #3:** Here's another: you could use **guard** or **precondition()** to ensure both numbers are positive.

## Solution

This ought to have been the easiest of easy challenges, because all you're doing is multiplying a number against itself a fixed number of times.

## Challenge 18: Recreate the pow() function

In its most simple form, you can solve the challenge like this:

```
func challenge18a(number: Int, power: Int) -> Int {  
    guard number > 0, power > 0 else { return 0 }  
    var returnValue = number  
  
    for _ in 1..        returnValue *= number  
    }  
  
    return returnValue  
}
```

Like I said in the hints, you could also solve this challenge using a recursive function. To do that, make the function multiply its input number by the return value of calling itself, subtracting 1 from the power parameter each time, like this:

```
func challenge18b(number: Int, power: Int) -> Int {  
    guard number > 0, power > 0 else { return 0 }  
    if power == 1 { return number }  
  
    return number * challenge18b(number: number, power: power -  
    1)  
}
```

# Challenge 19: Swap two numbers

**Difficulty:** Easy

Swap two positive variable integers, **a** and **b**, without using a temporary variable.

## Sample input and output

- Before running your code **a** should be 1 and **b** should be 2; afterwards, **b** should be 1 and **a** should be 2.

## Hints

**Hint #1:** There are lots of ways to solve this, but probably the easiest to remember is using tuples.

**Hint #2:** Alternatively, try using the global Swift function `swap()`.

**Hint #3:** If you're feeling fancy, you can solve this problem with arithmetic.

**Hint #4:** If you're feeling fancy and want to demonstrate your bit manipulation skills, you can also solve this problem using bitwise XOR.

## Solution

This is a favorite question of lazy interviewers – people who don't want to spend an hour of life coming up with genuinely interesting, useful questions that explore real Swift knowledge. It *used* to be important in Ye Olde Days when every byte mattered, but in a world where Slack on macOS happily chews through 400MB of RAM just idling this test is more a curiosity than anything else.

Still, this question does one have benefit, which is that are idiomatic solutions for Swift

## Challenge 19: Swap two numbers

developers – i.e., Swift ways to solve it.

Let's look at the basic solution first, which looks like this:

```
a = a + b  
b = a - b  
a = a - b
```

That's the solution you'd use in most languages, and it works fine. If you're feeling smart, you can also use XOR like this:

```
a = a ^ b  
b = a ^ b  
a = a ^ b
```

I think that solution is fractionally easier to remember, because it uses the same operator all three times.

So, those are the standard solutions, but Swift gives us two alternatives. First, there's a global **swap()** function that swaps two values of the same type, like this:

```
swap(&a, &b)
```

The **swap()** function is micro-optimized to be as fast as possible, so you'll see it used extensively in sorting algorithms.

A second idiomatic solution is to use tuples. This delivers a beautifully simple solution to the challenge that is also undeniably Swiftly:

```
(a, b) = (b, a)
```

Marvelous.

# Challenge 20: Number is prime

**Difficulty:** Tricky

Write a function that accepts an integer as its parameter and returns true if the number is prime.

**Tip:** A number is considered prime if it is greater than one and has no positive divisors other than 1 and itself.

## Sample input and output

- The number 11 should return true.
- The number 13 should return true.
- The number 4 should return false.
- The number 9 should return false.
- The number 16777259 should return true.

## Hints

**Hint #1:** You should start with a brute force approach: loop through every number from 2 up to one less than the input number, and check whether the input number divides into it.

**Hint #2:** You can shrink the search space by searching up to a smaller number – what's the highest it could be?

**Hint #3:** There's no point searching higher than the square root of your input number, rounding up.

## Solution

There's a naïve solution to this problem, but it has terrible performance characteristics – there's a reason I included 16,777,259 in the list of sample input and output.

The naïve solution looks like this:

```
func challenge20a(number: Int) -> Bool {
    guard number >= 2 else { return false }

    for i in 2 ..< number {
        if number % i == 0 {
            return false
        }
    }

    return true
}
```

That counts from 2 up to one less than the input number, and returns false if the input number divides equally into **i**. That works correctly. It has a **guard** statement at the front because the numbers 1 and lower are not prime by definition.

The problem with this function is that it's computationally expensive: 16,777,259 is a prime number, so this solution will divide from 2 up to 16,777,258 and find that none of them work before deciding that the number is prime.

Consider this: if the number **n** is not prime, it means it can be reached by multiplying two factors, **x** and **y**. If both of those numbers were greater than the square root of **n**, then **x \* y** would be greater than **n**, which is not possible. So, we can be sure that at least one of **x** or **y** is less than or equal to the square root of **n**.

As a result of this, we can dramatically reduce our search space: rather than counting from 2 up to 16,777,259, we can square root the number and round up, to get 4097, then search up to there and no further. Remember, we don't have to find both numbers that multiply to make **n**, just one of them, because if we find one – and it isn't 1 or itself – it means **n** is not prime.

So, we could write a second solution like this:

## Numbers

```
func challenge20b(number: Int) -> Bool {  
    guard number >= 2 else { return false }  
    guard number != 2 else { return true }  
    let max = Int(ceil(sqrt(Double(number))))  
  
    for i in 2 ... max {  
        if number % i == 0 {  
            return false  
        }  
    }  
  
    return true  
}
```

Because this second solution uses the closed range operator, `...`, rather than the half-open range operator, `.., it's important to add the second guard check at the top so that 2 doesn't evaluate incorrectly.`

This second solution performs significantly faster for primes such as 16,777,259, because rather than around 16 million searches you're now doing around 4000.

# Challenge 21: Counting binary ones

**Difficulty:** Tricky

Create a function that accepts any positive integer and returns the next highest and next lowest number that has the same number of ones in its binary representation. If either number is not possible, return nil for it.

## Sample input and output

- The number 12 is 1100 in binary, so it has two 1s. The next highest number with that many 1s is 17, which is 10001. The next lowest is 10, which is 1010.
- The number 28 is 11100 in binary, so it has three 1s. The next highest number with that many 1s is 35, which is 100011. The next lowest is 26, which is 11010.

## Hints

**Hint #1:** You can find the binary representation of an integer by converting it to a string – look for a “radix” initializer.

**Hint #2:** You should be using radix 2, which is binary.

**Hint #3:** Your return value ought to be (**nextHighest: Int?, nextLowest: Int?**).

**Hint #4:** You can count the 1s in a stringified number by using **filter()** on its letters property.

**Hint #5:** Don’t be afraid to duplicate code while you’re working – you need to search up and down for the same thing, so start with duplication then refactor.

**Hint #6:** You can’t create ranges where the end is higher than the start. Instead, create a forwards range then reverse it.

## Solution

This is a classic computing science problem, although I have to admit Swift makes it quite easy thanks to its range of **String** initializers.

First and most importantly, you get the binary representation of an integer like this:

```
let binaryString = String(someNumber, radix: 2)
```

With that done, you can count the 1s by filtering by character and counting the resulting array:

```
let numberOfOnes = binaryString.filter { (char: Character) ->
    Bool in char == "1" }.count
```

All that leaves is finding the next highest and lowest, which can be done by counting upwards or downwards until you back the same number of ones.

Here's the complete solution:

```
func challenge21a(number: Int) -> (nextHighest: Int?,  
nextLowest: Int?) {  
    let targetBinary = String(number, radix: 2)  
    let targetOnes = targetBinary.filter { (char: Character) ->  
        Bool in char == "1" }.count  
  
    var nextHighest: Int? = nil  
    var nextLowest: Int? = nil  
  
    for i in number + 1...Int.max {  
        let currentBinary = String(i, radix: 2)  
        let currentOnes = currentBinary.filter { (char:  
Character) -> Bool in char == "1" }.count
```

## Challenge 21: Counting binary ones

```
        if targetOnes == currentOnes {
            nextHighest = i
            break
        }
    }

    for i in (0 ..< number).reversed() {
        let currentBinary = String(i, radix: 2)
        let currentOnes = currentBinary.filter { (char: Character) -> Bool in char == "1" }.count

        if targetOnes == currentOnes {
            nextLowest = i
            break
        }
    }

    return (nextHighest, nextLowest)
}
```

Looking at that code, the duplication of binary counting does rather stick out. We can refactor it into a nested function to remove the duplication, although it only reduces the overall size of the solution by a little:

```
func challenge21b(number: Int) -> (nextHighest: Int?,  
nextLowest: Int?) {
    func ones(in number: Int) -> Int {
        let currentBinary = String(number, radix: 2)
        return currentBinary.filter { (char: Character) -> Bool  
in char == "1" }.count
    }
}
```

## Numbers

```
let targetOnes = ones(in: number)
var nextHighest: Int? = nil
var nextLowest: Int? = nil

for i in number + 1...Int.max {
    if ones(in: i) == targetOnes {
        nextHighest = i
        break
    }
}

for i in (0 ..< number).reversed() {
    if ones(in: i) == targetOnes {
        nextLowest = i
        break
    }
}

return (nextHighest, nextLowest)
}
```

# Challenge 22: Binary reverse

**Difficulty:** Tricky

Create a function that accepts an unsigned 8-bit integer and returns its binary reverse, padded so that it holds precisely eight binary digits.

**Tip:** When you get the binary representation of a number, Swift will always use as few bits as possible – make sure you pad to eight binary digits before reversing.

## Sample input and output

- The number 32 is 100000 in binary, and padded to eight binary digits that's 00100000. Reversing that binary sequence gives 00000100, which is 4. So, when given the input 32 your function should return 4.
- The number 41 is 101001 in binary, and padded to eight binary digits that 00101001. Reversing that binary sequence gives 10010100, which is 148. So, when given the input 41 your function should return 148.
- It should go without saying that your function should be symmetrical: when fed 4 it should return 32, and when fed 148 it should return 41.

## Hints

**Hint #1:** You can get the binary representation of an integer using `String(someNumber, radix: 2)`.

**Hint #2:** You can get the decimal integer equivalent of a string containing binary using `Int(someString, radix: 2)` – but be warned that will give you an optional integer.

**Hint #3:** To pad the input number's binary representation so that it holds eight digits, use the `String(repeating:count:)` initializer.

**Hint #4:** You can reverse a character array then create a new string from it.

## Solution

As long as you're comfortable converting to and from binary, this challenge is not likely to pose any problems for you. In fact, apart from converting between binary and decimal, the only interesting parts of the problem are calculating how much zero padding to apply and reversing the binary representation.

Calculating and adding the padding can be done by subtracting the current character count from 8, like this:

```
let paddingAmount = 8 - binary.count  
let paddedBinary = String(repeating: "0", count: paddingAmount)  
+ binary
```

Reversing the padding binary representation is as easy as calling `reversed()` on the binary string, then creating a new string out of it, like this:

```
let reversedBinary = String(paddedBinary.reversed())
```

With all that in mind, here's a complete solution:

```
func challenge22(number: UInt) -> UInt {  
    let binary = String(number, radix: 2)  
    let paddingAmount = 8 - binary.count  
    let paddedBinary = String(repeating: "0", count:  
paddingAmount) + binary  
    let reversedBinary = String(paddedBinary.reversed())  
    return UInt(reversedBinary, radix: 2)!  
}
```

# Challenge 23: Integer disguised as string

**Difficulty:** Tricky

Write a function that accepts a string and returns true if it contains only numbers, i.e. the digits 0 through 9.

## Sample input and output

- The input “01010101” should return true.
- The input “123456789” should return true.
- The letter “9223372036854775808” should return true.
- The letter “1.01” should return false; “.” is not a number.

## Hints

**Hint #1:** You can create integers from strings, and Swift will return nil if the conversion failed.

**Hint #2:** The number “9223372036854775808” is a precise choice, not just a random string of numbers.

**Hint #3:** Chances are Swift’s integer type will be 64-bit for you, and it’s signed, which means its maximum value is 2 to the power of 63, minus 1, i.e. 9223372036854775807 – that’s one less than the test case you’ve been given.

**Hint #4:** You should look into inverted character sets.

**Hint #5:** Some languages write numbers differently from English.

## Numbers

### Solution

There are lots of ways of solving this challenge, but if you want to solve it clearly, concisely, and correctly then your options are more limited.

There are two big gotchas when dealing with numbers. First, integers have a ceiling, beyond which they refuse to work. In Swift, the ceiling is 9,223,372,036,854,775,807, which is the largest number that can be represented by a signed 64-bit integer.

The third test case I gave you was one higher than the maximum signed 64-bit integer, which was intentional. However, if you recognized that, you could have switched an unsigned integer and passed the challenge by writing code like this:

```
func challenge23a(input: String) -> Bool {
    return UInt(input) != nil
}
```

I can't think of a faster, simpler way to solve this challenge. However, it's also a bit of a fudge – all unsigned integers do is double the largest number you can address, so it would still fail if you added a 0 to the end of the existing big number.

From there you might have migrated to using `Int()` to compare each individual letter in the string, like this:

```
func challenge23b(input: String) -> Bool {
    for letter in input {
        if Int(String(letter)) == nil {
            return false
        }
    }

    return true
}
```

## Challenge 23: Integer disguised as string

As you can see, you can create integers from strings, and strings from characters, but you can't create integers from characters – d'oh. Still, though, this code runs fast: it bails out as soon as any non-matching letter is found, and only returns true if all digits converted to an integer successfully.

An alternative solution is to use the **rangeOfCharacter(from:)** method, which lets you provide a character set and returns the location – if any – of those characters in the search string. In our case we know the numbers we want (digits), so we just need to get the inverse of that set using something like this:

```
func challenge23c(input: String) -> Bool {
    return input.rangeOfCharacter(from:
CharacterSet.decimalDigits.inverted) == nil
}
```

This solution is interesting because it highlights another curiosity of numbers: Apple, being the smart company it is, considers “decimal digits” to include numerals from other languages. I'm not sure how well this will print on your computer, but “ߵ” is the Arabic-Indic numeral 2.

If you use the **decimalDigits** character set, it will include Arabic-Indic numerals as well as the numbers 0 through 9. While none of the test cases used Arabic-Indic numerals, if you wanted to conform strictly to the requirements for this challenge then you could use something like this:

```
func challenge23d(input: String) -> Bool {
    return input.rangeOfCharacter(from:
CharacterSet(charactersIn: "0123456789").inverted) == nil
}
```

Perhaps now you understand why I graded this challenge as tricky rather than easy!

# Challenge 24: Add numbers inside a string

**Difficulty:** Tricky

Given a string that contains both letters and numbers, write a function that pulls out all the numbers then returns their sum.

## Sample input and output

- The string “a1b2c3” should return 6 ( $1 + 2 + 3$ ).
- The string “a10b20c30” should return 60 ( $10 + 20 + 30$ ).
- The string “h8ers” should return “8”.

## Hints

**Hint #1:** Creating an integer from a string returns **Int?** – nil if it was a number, or an **Int** otherwise.

**Hint #2:** Use the nil coalescing operator, **??**, to strip out any unwanted optionality.

**Hint #3:** Don’t forget to catch trailing numbers, i.e. where the string ends with a number.

**Hint #4:** You could solve this using regular expressions, in which case I’d grade it as taxing rather than tricky.

## Solution

This is a personal favorite problem of mine – not because of its complexity, but more because of its usefulness for weeding out developers who have exaggerated a little on their résumé.

## Challenge 24: Add numbers inside a string

You see, you can take a naïve approach to this challenge and get a solution that works efficiently with very little code. However, to do that you need to know how to convert characters to strings, and strings to integers, as well as how to use nil coalescing if you want to make the code neat. As a result, someone who has puffed up their résumé will find this harder than it ought to be, and their code will usually show them up.

On the flip side, an experienced Swift developer who is out to impress might try and solve this using regular expressions, in which case now they have two challenges: summing the numbers, and making regular expressions work under pressure.

Let's take a look at the simple solution first, which is hopefully similar to the one you came up with. The algorithm is this:

1. Create an empty string that represents the current number being read.
2. Create a **sum** value that contains the total of all numbers so far, initialized to 0.
3. Loop through every letter in the input string, converting the character to a **String**.
4. If we can convert that string to an integer, add it to the current number string.
5. Otherwise it's not a number, so convert the current number string to an integer, or 0 if it's an invalid integer, add it to the **sum** and clear the current number string.
6. Finally, convert any remaining value in the current number string to an integer, and add it to **sum**.
7. Return **sum**.

Here's that list translated into code:

```
func challenge24a(string: String) -> Int {  
    var currentNumber = ""  
    var sum = 0  
  
    for letter in string {  
        let strLetter = String(letter)  
  
        if Int(strLetter) != nil {  
            currentNumber += strLetter  
        } else {  
            if currentNumber != "" {  
                sum += Int(currentNumber) ?? 0  
                currentNumber = ""  
            }  
        }  
    }  
    if currentNumber != "" {  
        sum += Int(currentNumber) ?? 0  
    }  
    return sum  
}
```

## Numbers

```
        currentNumber += strLetter
    } else {
        sum += Int(currentNumber) ?? 0
        currentNumber = ""
    }
}

sum += Int(currentNumber) ?? 0
return sum
}
```

Solving the same problem using regular expressions is a real nightmare, and highlights the worst of Swift's string handling problems. You see, creating a regex requires a Swift string going in, but works entirely using **NSRange** rather than Swift's string ranges. This means you need to use **string.utf16.count** to calculate the size of the range, and using **string.count** will introduce subtle bugs in your code.

However, it gets really grim when trying to read the contents of each match.

**NSRegularExpression** returns an array of **NSTextCheckingResult**, which contains the **NSRange** of each match but not the contents, so you need to read the substring using that range... which isn't possible in Swift, because it uses string ranges rather than **NSRange**.  
*\*Sigh\**...

Anyway, if you want to prove you have more time than sense, here's how to solve it using regular expressions:

1. Create the regex `(\d+)`.
2. Use its **matches()** method to pull out an array of **NSTextCheckingResult** objects.
3. Typecast your input string as **NSString** to get a **substring()** method that works with the **NSRange** provided by each **NSTextCheckingResult**.
4. Send that substring into **Int()** to get an optional integer.
5. Strip out the optionality, then sum the integers.

## Challenge 24: Add numbers inside a string

You can use **compactMap()** and **reduce()** for steps 4 and 5 if you really have something to prove, giving code like this:

```
func challenge24b(string: String) -> Int {
    let regex = try! NSRegularExpression(pattern: "(\\d+)",
options: [])
    let matches = regex.matches(in: string, options: [], range:
NSRange(location: 0, length: string.utf16.count))

    let allNumbers = matches.compactMap { Int((string as
NSString).substring(with: $0.range)) }

    return allNumbers.reduce(0, +)
}
```

That code runs *significantly* slower than the previous solution, largely because of the cost of creating the regex. If you were to run this method many times you should move the **NSRegularExpression** creation outside the method, at which point it runs “only” about half the speed of the previous solution.

An alternative regex solution is to use Swift’s built-in regular expression replacement code to replace all non-numbers with something unique, such as dashes. We can then split by that character to make an array, then reduce that down to a single value, like this:

```
func challenge24c(string: String) -> Int {
    return string
        .replacingOccurrences(of: "\\D+", with: "-",
options: .regularExpression)
        .split(separator: "-")
        .reduce(0) { $0 + Int(String($1))!
}
```

If regex isn’t your thing, **\d** means “anything that is a number” and **\D** means “anything that

## Numbers

*isn't* a number. So, that will replace letters and punctuation with dashes, before splitting on dashes and reducing the resulting array down to a single value.

# Challenge 25: Calculate a square root by hand

**Difficulty:** Taxing

Write a function that returns the square root of a positive integer, rounded down to the nearest integer, without using `sqrt()`.

## Sample input and output

- The number 9 should return 3.
- The number 16777216 should return 4096.
- The number 16 should return 4.
- The number 15 should return 3.

## Hints

**Hint #1:** You can brute force this using a loop count from 0 up to the input number.

**Hint #2:** A more efficient solution is using a binary search.

**Hint #3:** A rounded-down integer square root will never be more than half its square.

**Hint #4:** If you consider half your input number to be your upper bound, then calculate the mid-point between that and a lower bound that's initially 0 (i.e., input number / 4), you can check whether that mid-point squared gives your input.

**Hint #5:** If the mid-point is too low, make it the new lower bound then repeat. If the mid-point is too high, make it the new higher bound then repeat.

**Hint #6:** Using this technique, you should be able to loop until you find the best answer.

## Solution

Like most coding interview problems, this one has a naïve solution, a smart solution, and a sneaky solution.

The naïve solution is trivial: loop from 1 up to one higher than half the test number, checking to see whether that number squared is greater than the test number. If it is, return the number directly below. We need to add a special case for 1, because otherwise the code returns 0.

Here's the code:

```
func challenge25a(input: Int) -> Int {  
    guard input != 1 else { return 1 }  
  
    for i in 0 ... input / 2 + 1 {  
        if i * i > input {  
            return i - 1  
        }  
    }  
  
    return 0  
}
```

However, consider the second test case, which is calculating the square root of 16,777,216. That's going to require 4097 multiplications before returning the correct response, which is massively wasteful.

A smarter solution is to use a binary search, which massively reduces the search space. Given the input number 10 it works like this:

- Start by specifying a high bound of half the input number plus one, rounding down so for the input number 10 that gives us a high bound of 6.
- Now specify a low bound, which will be zero to begin with.

## Challenge 25: Calculate a square root by hand

- Calculate the mid-point of the two, which is equal to half of **upper - lower**, plus the lower. So that's **lower + ((upper - lower) / 2)**, which is **0 + ((6 - 0) / 2)**, which is **0 + (6 / 2)**, which is **0 + 3** – so our mid-point is 3.
- You then square your mid-point (3 x 3 is 9) and compare it against the input number.
- If the square is less than the input it means that all the numbers from your low bound up to the mid-point are also too low, and so don't need to be checked, so you can set the new low bound to be equal to your mid-point, then repeat.
- If the square is higher than the input it means that all the numbers from the mid-point up to the upper bound are also too high, and so don't need to be checked, so you can set the new high bound to be equal to your mid-point, then repeat.
- If the square is equal to the input, then you have your answer.
- If you find that your lower bound + 1 is greater than or equal to your upper bound it means you've overshot the mark, so you should return the lower bound.

I realize that sounds complicated, but it's actually remarkably simple: “select the range it must be in, then try the middle of it. If you were too high you can eliminate the upper half of the range; if you were too low you can eliminate the lower half of the range. So, eliminate one half of the range then split the remaining half... and repeat.”

This search technique is sometimes called a binary chop, because you halve your search space with each check. So to get 16,777,216 you halve to a range of 8,388,608, then halve that to a range of 4,194,304, then halve to a range of 2,097,152, then 1,048,576, then 524,288, then 262,144, then 131,072, then 65,536, and so on.

Using this approach takes only 11 loops to figure out the square root of 16,777,216, compared to 4097 loops using the naïve method, so it runs a great deal faster. Nice! Here's the code:

```
func challenge25b(input: Int) -> Int {  
    guard input != 1 else { return 1 }  
  
    var lowerBound = 0  
    var upperBound = 1 + input / 2
```

## Numbers

```
while lowerBound + 1 < upperBound {  
    let middle = lowerBound + ((upperBound - lowerBound) / 2)  
    let middleSquared = middle * middle  
  
    if middleSquared == input {  
        return middle  
    } else if middleSquared < input {  
        lowerBound = middle  
    } else {  
        upperBound = middle  
    }  
}  
  
return lowerBound  
}
```

So, that's the naïve approach and the smart approach, but there's also a sneaky approach: the challenge is to calculate the square root of an integer without using `sqrt()`, but it *didn't* say not to use the `pow()` function. If you request a number raised to the power of 0.5, you get its square root. The calculation isn't precisely the same – a true square root will yield ever so slightly different results, and will be optimized for that task – but given that we're working with integers the two will be identical.

So, here's how to solve the challenge using `pow()`, which will run so fast it's hard to benchmark meaningfully:

```
func challenge25c(input: Int) -> Int {  
    return Int(floor(pow(Double(input), 0.5)))  
}
```

Note the extensive typecasting, which is unavoidable I'm afraid – `pow()` works with doubles, not integers, so we need to typecast and floor before converting to an integer for the return value.

Challenge 25: Calculate a square root by hand

# Challenge 26: Subtract without subtract

**Difficulty:** Taxing

Create a function that subtracts one positive integer from another, without using `-`.

## Sample input and output

- The code `challenge26(subtract: 5, from: 9)` should return 4.
- The code `challenge26(subtract: 10, from: 30)` should return 20.

## Hints

**Hint #1:** In your code you can use any other operator, or any other number, positive or negative, just not the `-` operator.

**Hint #2:** Swift has a full set of bitwise operators – operators that manipulate the binary digits of a number.

**Hint #3:** You could try using bitwise NOT, which is `~`.

## Solution

This question is looking for a basic grasp of mathematics: if you can't subtract one number from another using `-`, how can you do it? The answer is, of course, to flipping the sign on the number then adding it – i.e., rather than subtracting 10, you add -10.

You could make an argument that flipping the sign on a number using `-` is different to using `-` for subtraction. That would give you a solution like this:

## Challenge 26: Subtract without subtract

```
func challenge26a(subtract: Int, from: Int) -> Int {  
    return from + -subtract  
}
```

Technically, though, `-` in that instance is the unary minus operator, so it's the same thing, really.

A solution that fits the spirit of the challenge a bit better is multiplying by `-1`, like this:

```
func challenge26b(subtract: Int, from: Int) -> Int {  
    return from + -1 * subtract  
}
```

That's a negative number, not the unary minus operator, so that fits both the letter and spirit of the challenge.

However, for maximum effect, you can solve this challenge without typing `-` at all – well, excluding the `->` used in declaring the function's return value.

This technique depends on the `~` operator (a tilde), which is bitwise NOT. It causes all the binary digits in a number to be flipped. If this is not new to you, skip ahead – I'm going to take a brief tangent into what it does behind the scenes.

8-bit integers are stored using eight binary digits, where the right-most digit stores 1, the second right-most stores 2, then 4, 8, 16, 32, and 64. Depending on whether each of those binary digits are 1 or 0, Swift can represent numbers from 0 (all zeros) to 127 (all 1s).

However, notice there are only seven of them – 1, 2, 4, 8, 16, 32, and 64 – and we're talking about 8-bit integers, so there ought to be 8, right? Right.

That eighth digit is how we track whether a number is positive or negative. If the left-most digit is 0, all the digits on the right add up to a positive number. If the left-most digit is 1, all the digits on the right represent a negative number.

Here are some examples of positive numbers:

## Numbers

- 00000000 is 0
- 00000001 is 1
- 00000010 is 2
- 00000011 is 3
- 00000100 is 4
- 00000101 is 5
- 01000000 is 64
- 01111111 is 127

So, as long as that first digit is 0, the rest of the digits form a positive number.

Things are a little tricksier when you use negative numbers. Whereas the highest positive number starts with a zero and is followed by all ones, the lowest negative number starts with a one and is followed by all zeros – it's the exact opposite.

So:

- 00000000 is still 0
- 10000000 is -128
- 10000001 is -127
- 10000010 is -126
- 10000011 is -125
- 10000100 is -124
- 10000101 is -123
- 11000000 is -64
- 11111111 is -1

Now, all this becomes important when you flip the 1s and 0s. For example, 01000000 is 64, but if you make the 0s into 1s and the 1s into 0s you get 10111111, which is -65. Similarly, if you take 01010101, which is 85, and flip the bits, you get 10101010, which is -86.

Both times the negative number is identical to add one to the number and flipping its sign, i.e. making positive negative. And that's where our solution comes in – check this out:

## Challenge 26: Subtract without subtract

```
func challenge26c(subtract: Int, from: Int) -> Int {  
    return from + (~subtract + 1)  
}
```

So, to subtract one number from another, we flip the bits (64 becomes -65) then add one (to make -64), and add that to our input number to make subtraction. Done!

# **Chapter 3**

## Files

# Challenge 27: Print last lines

**Difficulty:** Easy

Write a function that accepts a filename on disk, then prints its last  $N$  lines in reverse order, all on a single line separated by commas.

## Sample input and output

Here is your test input file:

```
Antony And Cleopatra
Coriolanus
Cymbeline
Hamlet
Julius Caesar
King Lear
Macbeth
Othello
Twelfth Night
```

- If asked to print the last 3 lines, your code should output “Twelfth Night, Othello, Macbeth”.
- If asked to print the last 100 lines, your code should output “Twelfth Night, Othello, Macbeth, King Lear, Julius Caesar, Hamlet, Cymbeline, Coriolanus, Antony and Cleopatra”.
- If asked to print the last 0 lines, your could should print nothing.

## Hints

**Hint #1:** Use the `contentsOfFile` initializer to pull in the text, then `components(separatedBy:)` to create an array of lines.

## Files

**Hint #2:** Arrays have a built-in **reverse()** method that flip them around in-place.

**Hint #3:** You need to print the last  $N$  lines, but of course you don't want to read beyond the size of the array. Make sure you use the **min()** function to choose the lesser of the two.

## Solution

This is a nice and simple challenge, and is the kind of thing you'll hit in real-world coding all the time. Hopefully, then, you found it a breeze: use **contentsOfFile** to get a multiline **String** of text, use **components(separatedBy: "\n")** to split it into an array of lines, then reverse the array and loop over  $N$  items.

In order to fully satisfy the challenge's requirements – and to add safety to make sure we don't read outside the array! – we also need to add a couple of **guard** statements to eliminate bad input, and use **min()** to pick the lowest number between the requested line count and the number of lines in the array.

Here's the code:

```
func challenge27(filename: String, lineCount: Int) {  
    guard let input = try? String(contentsOfFile: filename) else  
{ return }  
  
    var lines = input.components(separatedBy: "\n")  
    guard lines.count > 0 else { return }  
  
    lines.reverse()  
  
    var output = [String]()  
  
    for i in 0 ..< min(lines.count, lineCount) {  
        output.append(lines[i])  
    }  
}
```

## Challenge 27: Print last lines

```
}

print(output.joined(separator: " , "))

}
```

# Challenge 28: Log a message

**Difficulty:** Easy

Write a logging function that accepts a path to a log file on disk as well as a new log message. Your function should open the log file (or create it if it does not already exist), then append the new message to the log along with the current time and date.

**Tip:** It's important that you add line breaks along with each message, otherwise the log will just become jumbled.

## Sample input and output

- If the file does not exist, running your function should create it and save the new message.
- If it does exist, running your function should load the existing content and append the message to the end, along with suitable line breaking.

## Hints

**Hint #1:** I think it would be reasonable to use `contentsOfFile` to load the existing log file into a string.

**Hint #2:** You can use `try?` and nil coalescing to provide a default value if the existing log file doesn't exist.

**Hint #3:** How you format the date and time for each message will be interesting, but don't forget KISS: Keep It Simple, Stupid.

**Hint #4:** What should happen if you can't write the log file? This wasn't specified in the challenge description, so you get to use some initiative.

## Solution

This is a small and contained challenge, but gives you just enough scope to demonstrate your Swift skills.

Specifically, there are three areas where your solution is open for discussion:

1. How you load the log file, or provide a default value if the log hasn't been created yet.
2. How you format dates to include in each log message.
3. How you write the updated log file back to disk, handling any errors that might occur.

The easiest way to solve the first is using one of my favorite Swift tips: combining `try?` with `nil` coalescing to provide a default value in cases where a thrown exception just means “missing value”. In code it looks like this:

```
var existingLog = (try? String(contentsOfFile: logFile)) ?? ""
```

After that line has run, `existingLog` will either be an empty string or the contents of the log file.

There are lots of ways to solve the second, but honestly the smartest way is also the easiest: just using `Date()` inside string interpolation. That will cause the current date and time to be printed out as “Year-Month-Day Hour:Minute:Second”, which is ideal. So, you would write this:

```
existingLog.append("\(Date()): \(message)\n")
```

Finally, writing data to disk is a throwing function, so you need to decide what to do with any errors. You *could* ignore them, like this:

```
_ = try? existingLog.write(toFile: logFile, atomically: true,
encoding: .utf8)
```

...but that's probably not smart: failure to write log messages seems serious to me, so at the very least you will want to print a warning, like this:

## Files

```
do {
    try existingLog.write(toFile: logFile, atomically: true,
encoding: .utf8)
} catch {
    print("Failed to write to log: \
(error.localizedDescription)")
}
```

Another reasonable approach would be to do without **do/catch**, just use **try** by itself, then mark the whole function with **throws** and let the call site deal with it. Regardless of what you choose, it leaves some scope for interesting discussion, which is always a good thing.

Here's my complete solution to this challenge:

```
func challenge28(log message: String, to logFile: String) {
    var existingLog = (try? String(contentsOfFile: logFile)) ??
    ""

    existingLog.append("\(Date()): \(message)\n")

    do {
        try existingLog.write(toFile: logFile, atomically: true,
encoding: .utf8)
    } catch {
        print("Failed to write to log: \
(error.localizedDescription)")
    }
}
```

# Challenge 29: Documents directory

**Difficulty:** Easy

Write a function that returns a **URL** to the user's documents directory.

## Sample input and output

- Your function should need no input, and return a URL pointing to /Users/yourUserName/Documents on macOS, and /path/to/container/Documents on iOS.

## Hints

**Hint #1:** This is one you either know or you don't. I'd be tempted to answer "that's something I could find on Google" if the answer fled from my brain in an interview.

**Hint #2:** You should investigate the **urls(for:in)** method of **FileManager**.

**Hint #3:** The user has only one documents directory.

## Solution

This is the kind of question that gives coding interviews a bad rap: there's no logic that you can figure out yourself, or any algorithm you might have learned in a class once. Instead, you either know the answer or you don't, which is testing nothing other than your ability to memorize vast swathes of Cocoa Touch.

There are three reasons I'm including it here. First, if you didn't know how to do this, you do know now so you've learned something new. Second, it's actually pretty easy to do once you know how, so you might as well get it under your belt in case you get hit with this question in

## Files

the future. Third, this is useful code to have at hand: reading and writing to the user's documents directory is very common on iOS, so you'll almost certainly use this code in real projects.

Here's my solution:

```
func challenge29() -> URL {
    let paths =
FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask)
    return paths[0]
}
```

Like I said in the hints, though: if you get asked this during an interview, I'd be tempted to answer that it's something trivial enough to be looked up online. I'm no fan of coding by Stack Overflow, and if someone said "I don't know much about **UITableView** but I could look it up" then I'd not think highly of their skills, but this particular problem is something so simple that it really is just a Google search away if you need it.

# Challenge 30: New JPEGs

**Difficulty:** Easy

Write a function that accepts a path to a directory and returns an array of all JPEGs that have been created in the last 48 hours.

**Tip #1:** For the purpose of this task, just looking for “.jpg” and “.jpeg” file extensions is sufficient.

**Tip #2:** For this challenge, assume time is regular and constant, i.e. the user has not changed their timezone or moved into or out from daylight savings.

**Tip #3:** Use the terminal command **touch -t YYYYMMDDHHMM somefile.jpg** to adjust the creation time of a file, e.g. **touch -t 201612250101**.

## Sample input and output

- If your directory contains three JPEGs older than 48 hours and three newer, your function should return the names of the three newer.

## Hints

**Hint #1:** Use the **contentsOfDirectory()** method of **FileManager** to pull out the list of files in your target directory.

**Hint #2:** You can compare two dates directly using **>**.

**Hint #3:** You can read the attributes of a file using the **attributesOfItem()** method on **FileManager**, and you’ll want to read the **FileAttributeKey.creationDate** from that dictionary.

## Solution

This challenge tests two things at once: do you feel comfortable working with files, and do you know how to create and compare dates? Neither of these are difficult, but they can show a lack of experience if you're hit with a question like this during an interview.

There are three parts to solving this challenge:

1. Getting all files in the target directory, using the **contentsOfDirectory()** method on **FileManager**.
2. Filtering by JPEGs. The challenge statement makes it clear that just filtering on the path extension of JPEG and JPG is sufficient, so you can just check the **pathExtension** property of each file.
3. Filtering by creation date, ensuring that only newer JPEGs are included.

That last part is the most complicated part of this problem, but even then it's not too hard. You can read all properties of a file, then pull out specifically its creation date, like this:

```
guard let attributes = try? fm.attributesOfItem(atPath:  
file.path) else { continue }  
guard let creationDate = attributes[.creationDate] as? Date  
else { continue }
```

Once you have the file's creation date, you can filter on files created after 48 hours ago by creating a new **Date** with the **timeIntervalSinceNow** initializer, passing in  $-60 * 60 * 48$  – i.e., 60 seconds \* 60 minutes \* 48 hours, with the minus in there to send it backwards in time from now:

```
if creationDate > Date(timeIntervalSinceNow: -60 * 60 * 48) {
```

Once you definitely have a valid file, you'll need to add it to an array then return that at the end of your function.

Here's my solution:

```
func challenge30(in directory: String) -> [String] {
    let fm = FileManager.default
    let directoryURL = URL(fileURLWithPath: directory)
    guard let files = try? fm.contentsOfDirectory(at: directoryURL, includingPropertiesForKeys: nil) else { return [] }

    var jpegs = [String]()

    for file in files {
        if file.pathExtension == "jpg" || file.pathExtension == "jpeg" {
            guard let attributes = try? fm.attributesOfItem(atPath: file.path) else { continue }
            guard let creationDate = attributes[.creationDate] as? Date else { continue }

            if creationDate > Date(timeIntervalSinceNow: -60 * 60 * 48) {
                jpegs.append(file.lastPathComponent)
            }
        }
    }

    return jpegs
}
```

# Challenge 31: Copy recursively

**Difficulty:** Easy

Write a function that accepts two paths: the first should point to a directory to copy, and the second should be where the directory should be copied to. Return true if the directory and all its contents were copied successfully; false if the copy failed, or the user specified something other than a directory.

## Sample input and output

- If your directory exists and is readable, the destination is writeable, and the copy succeeded, your function should return true.
- Under all other circumstances you should return false: if the directory does not exist or was not readable, if the destination was not writeable, if a file was specified rather than a directory, or if the copy failed for any other reason.

## Hints

**Hint #1:** **FileManager** has a dedicated **copyItem()** method that is perfectly capable of recursively copying directories.

**Hint #2:** To comply fully with the challenge, you must ensure the user does not specify a file to copy – this should accept only directories.

**Hint #3:** You should try the **fileExists()** method. It has a second parameter, specified as **inout**, that returns whether the requested item is a directory.

**Hint #4:** Never used **ObjCBool** before? Lucky you. Create your variable like this: **var isDirectory: ObjCBool = false**. Now call **fileExists()** then use its **boolValue** property to read the boolean value.

## Solution

On its surface this challenge needs you to know a little about the filesystem in Swift, in particular how much work **FileManager** can do for you.

However, you soon get hit with a cunning complexity: the **fileExists()** method returns true or false depending on whether the item exists, or also has an optional **inout** parameter that tells you whether the item was a directory. Cunningly, that **inout** parameter needs to be of type **ObjCBool**, which is like a regular boolean except that it's completely different and incompatible. Yay.

Fortunately, you can create an **ObjCBool** like this:

```
var isDirectory: ObjCBool = false
```

Once you pass that into **fileExists()**, it will be set to a new value that reflects the actual item you queried. You can then check whether it's a directory by reading the **boolValue** property of your **ObjCBool**, like this:

```
guard fm.fileExists(atPath: source, isDirectory: &isDirectory)
|| isDirectory.boolValue == false else { return false }
```

Now that you've checked the directory exists and that it is actually a directory, you can get onto the meat of this problem: copying it somewhere else. This is as simple as calling the **copyItem()** method of **FileManager**, although you will need to wrap it inside **do/catch** so you can catch any errors and return false, as the challenge requested.

Here's my solution:

```
func challenge31(source: String, destination: String) -> Bool {
    let fm = FileManager.default
    var isDirectory: ObjCBool = false

    guard fm.fileExists(atPath: source, isDirectory:
```

## Files

```
&isDirectory) || isDirectory.boolValue == false else { return  
false }  
  
let sourceURL = URL(fileURLWithPath: source)  
let destinationURL = URL(fileURLWithPath: destination)  
  
do {  
    try fm.copyItem(at: sourceURL, to: destinationURL)  
} catch {  
    print("Copy failed: \(error.localizedDescription)")  
    return false  
}  
  
return true  
}
```

# Challenge 32: Word frequency

**Difficulty:** Tricky

Write a function that accepts a filename on disk, loads it into a string, then returns the frequency of a word in that string, taking letter case into account. How you define “word” is worth considering carefully.

## Sample input and output

- A file containing “Hello, world!” should return 1 for “Hello”
- A file containing “Hello, world!” should return 0 for “Hello,” – note the comma at the end.
- A file containing “The rain in Spain falls mainly on the plain” should return 1 for Spain, and 1 for “in”; the “in” inside rain, Spain, mainly, and plain does not count because it’s not a word by itself.
- A file containing “I’m a great coder” should return 1 for “I’m”.

**Tip:** Create different files on your desktop for each of your pieces of sample input, then pass the paths to those files into your function.

## Hints

**Hint #1:** Being able to ask questions about definitions – “what is a word?” is an important skill in white boarding tests.

**Hint #2:** I would suggest that splitting by any non-alphabetic character is a safe choice for defining words to begin with, but watch out for that last test case.

**Hint #3:** There’s a built-in character set for letters, which includes uppercase and lowercase letters.

**Hint #4:** All character sets have an **inverted** property that gives you the opposite. For **letters** that gives you all non-letters.

## Files

**Hint #5:** Once you have a set of all non-letters, you can remove ' and split your string on that.

**Hint #6:** The **NSCountedSet** class can count words in an array extremely efficiently.

## Solution

There are two ways you could solve this, but both share a common component: getting an array of all the words. To do that, you should use **CharacterSet**, which lets you either construct a character set from scratch or use one of the built-in sets.

In this instance, the built-in **letters** character set is almost good enough: if we invert it, we'll get every non-letter. However, the last test case requires us to match "I'm" as one word, which means we can't split on apostrophes. So, the code looks like this:

```
var nonletters = CharacterSet.letters.inverted  
nonletters.remove(" ' ")  
  
let allWords = inputString.components(separatedBy: nonletters)
```

Now that you have an array of all the words, there are three ways you could perform your counting.

First, loop over every word then add 1 to its count, and finally put out the specific count for the word in question, like this:

```
var wordCounts = [String: Int]()  
  
for word in allWords {  
    wordCounts[word, default: 0] += 1  
}  
  
return wordCounts[count, default: 0] ?? 0
```

That code works, but it's not terribly efficient – while there are some situations where counting *every* word would be beneficial, it wasn't in the spec for this challenge. So, at the very least we could store significantly less data by just counting the one specific word we cared about, like this:

```
var wordCount = 0

for word in allWords {
    if word == count {
        wordCount += 1
    }
}

return wordCount
```

That's an improvement, and it certainly uses less memory now. But if you wanted to ace this challenge properly, the best solution of all is to use the Foundation type **NSCountedSet**. This is a specialized set that stores each item only once, so you get the performance of regular sets, but *acts* like items are stored more than once, so you can ask “how many times does this item appear?”

Using this, you can count all instances of a word like this:

```
let wordsSet = NSCountedSet(array: allWords)
    return wordsSet.count(for: count)
```

Regardless of which option you choose, the easy part is loading the file into a string – that can be done using the **contentsOfFile:** initializer for **String**, like this:

```
guard let inputString = try? String(contentsOfFile: filename)
else { return 0 }
```

## Files

That will either return the contents of the filename as a string, or return 0 immediately.

So, here's the final code for the function:

```
func challenge32(filename: String, count: String) -> Int {  
    guard let inputString = try? String(contentsOfFile:  
filename) else { return 0 }  
    var nonletters = CharacterSet.letters.inverted  
    nonletters.remove(" ")  
  
    let allWords = inputString.components(separatedBy:  
nonletters)  
    let wordsSet = NSCountedSet(array: allWords)  
  
    return wordsSet.count(for: count)  
}
```

As a side benefit, you also get the flexibility benefit from the dictionary code, in that if you need to query other words you can do so for free because all the work has been done up front.

# Challenge 33: Find duplicate filenames

**Difficulty:** Tricky

Write a function that accepts the name of a directory to scan, and returns an array of filenames that appear more than once in that directory or any of its subdirectories. Your function should scan *all* subdirectories, including subdirectories of subdirectories.

## Sample input and output

- If directory/subdir/a.txt exists and directory/sub/sub/sub/subdir/a.txt exists, “a.txt” should be in the array you return.
- Ignore directories that have the same name; you care only about files.
- If there are no files with duplicate names, return an empty array.

## Hints

**Hint #1:** There are several ways of solving this, but most people instinctively reach for recursion to allow them to scan any depth of directory structure.

**Hint #2:** **FileManager** has better solutions, though – take some time to explore!

**Hint #3:** **FileManager** has an uncomfortable relationship between using **String** and **URL** for its data types. You should prefer the latter wherever possible, however your return value needs to be **[String]** because you care about duplicate names not paths.

**Hint #4:** If you can’t read the directory that was requested, returning an empty array seems like a sensible thing to do.

## Files

### Solution

It's entirely possible you solved this challenge using a recursive function: return a list of all files in the current directory, along with the result of calling the function again on each subdirectory. That takes a little thinking, but it's an acceptable solution to this challenge.

There is, however, a better solution: **FileManager** can perform a deep search of directories for you. The best way to do this is with a directory enumerator, which you create like this:

```
guard let files = fm.enumerator(at: directoryURL,  
includingPropertiesForKeys: nil) else { return [] }
```

You can then loop over that enumerator and either get back a URL that points to a subdirectory or file, or you'll get back nil. The enumerator will automatically recurse into every subdirectory (and every sub-subdirectory, and so on), which takes care of most of this challenge for you.

One small fly in the ointment: looping over the iterator needs to be typecast to **URL**, so we'll be writing this:

```
for case let file as URL in files {
```

There are two things that remain. First, eliminating directories: we don't care if there are ten subdirectories called "test", we only care about duplicate filenames. So, whenever we're handed a URL that is a directory we are going to ignore it like this:

```
guard file.hasDirectoryPath == false else { continue }
```

The second remaining part of the challenge is to track duplicate filenames. We can solve this using two sets: one to track all filenames we've seen already, and one to track filenames that have been seen more than once. The challenge doesn't mention whether the return value should list duplicate filenames only once, but clearly it's more useful to do so, hence why I'm using a set.

Here's my solution in full:

## Challenge 33: Find duplicate filenames

```
func challenge33(in directory: String) -> [String] {
    let fm = FileManager.default
    let directoryURL = URL(fileURLWithPath: directory)

    guard let files = fm.enumerator(at: directoryURL,
includingPropertiesForKeys: nil) else { return [] }
    var duplicates = Set<String>()
    var seen = Set<String>()

    for case let file as URL in files {
        guard file.hasDirectoryPath == false else { continue }

        let filename = file.lastPathComponent

        if seen.contains(filename) {
            duplicates.insert(filename)
        }

        seen.insert(filename)
    }

    return Array(duplicates)
}
```

Recursion is always going to be necessary, but it's nice to have iOS do it for us!

# Challenge 34: Find executables

**Difficulty:** Tricky

Write a function that accepts the name of a directory to scan, and returns an array containing the name of any executable files in there.

## Sample input and output

- If directory/a exists and is executable, “a” should be in the array you return.
- If directory/subdirectory/a exists and is executable, it should *not* be in the array you return; only return files in the directory itself, not its subdirectories.
- If there are no executable files, return an empty array.

## Hints

**Hint #1:** Make sure you create the test directories as shown in the sample input/output. You can use **touch a** to create a file called “a”, then **chmod a+x a** to mark it executable.

**Hint #2:** You probably want to use the **isExecutableFile()** method of **FileManager**.

**Hint #3:** If you don’t create at least one subdirectory to test with, your tests will be incomplete.

**Hint #4:** Directories are considered executable for historical reasons. If they appear in your return array, you won’t have passed the challenge.

## Solution

This is an easy challenge to fail if you didn’t bother to create the sample input and check that your code works as expected. Working with filesystems is *always* replete with gotchas, and this challenge is no different: if you thought you could just get a list of all the files in the

directory and return those that are executable, your code won't work properly because **FileManager** considers directories to be executable.

The reason for this is historical: file permissions mean one thing when they are attached to files, and something else entirely when attached to directories. For files, executable permission means the file can be run, so it's a script or a binary. However, for directories executable permission means the user can enter into the directory and access its contents.

By default, all directories are considered executable, so you must explicitly filter out any directories you find. Once that's done, you can use **isExecutableFile()** to check what remains.

Here's my solution:

```
func challenge34(in directory: String) -> [String] {
    let fm = FileManager.default
    let directoryURL = URL(fileURLWithPath: directory)

    guard let files = try? fm.contentsOfDirectory(at:
        directoryURL, includingPropertiesForKeys: nil) else { return
    [] }

    var returnValue = [String]()

    for file in files {
        guard file.hasDirectoryPath == false else { continue }

        if fm.isExecutableFile(atPath: file.path) {
            returnValue.append(file.lastPathComponent)
        }
    }

    return returnValue
}
```

## Files

# Challenge 35: Convert images

**Difficulty:** Tricky

Write a function that accepts a path to a directory, then converts to PNGs any JPEGs it finds in there, leaving the originals intact. If any JPEG can't be converted the function should just quietly continue.

**Tip #1:** For the purpose of this task, just looking for “.jpg” and “.jpeg” file extensions is sufficient.

**Tip #2:** You can write this for iOS or macOS depending on your skills. iOS is easy to code but tricky to create a test environment for, and macOS is the opposite. If you can solve this on both platforms, I'd rate this Taxing.

## Sample input and output

- If your directory exists and is readable, all JPEGs in there should be converted to PNGs.
- If any JPEGs can't be read, converted, or written, just continue on quietly.

## Hints

**Hint #1:** If you're solving this on iOS, you will need to use `UIImage(contentsOfFile:)` and `pngData()`.

**Hint #2:** If you're solving this on macOS, you will need to use `NSImage` and `NSBitmapImageRep`.

**Hint #3:** Make sure and check the `pathExtension` property for each file.

**Hint #4:** Calling `write()` on a `Data` instance throws an exception if things go wrong, but the challenge wants you to ignore it and carry on – a perfect use for `try?`, but make sure you silence the warning.

## Solution

How you solve this depends on the platform you choose: there are *some* similarities, but just as many differences.

Let's take a look at iOS first, because it's the easiest option:

1. Get an array of all files in the target directory.
2. Loop over all files, ignoring any that don't end with "jpeg" or "jpg".
3. Create a **UIImage** out of the file, then use its **pngData()** method to create PNG data for it.
4. Create a new filename by replacing "jpg" with "png" in the filename.
5. Write the PNG data to disk using the new filename.

Here it is in code:

```
func challenge35(in directory: String) {
    let fm = FileManager.default
    let directoryURL = URL(fileURLWithPath: directory)
    guard let files = try? fm.contentsOfDirectory(at:
directoryURL, includingPropertiesForKeys: nil) else { return }

    for file in files {
        guard file.pathExtension == "jpeg" || file.pathExtension
== "jpg" else { continue }
        guard let image = UIImage(contentsOfFile: file.path) else
{ continue }
        guard let png = image.pngData() else { continue }

        let newFilename =
file.deletingPathExtension().appendingPathExtension("png")
        _ = try? png.write(to: newFilename)
    }
}
```

```
}
```

Accomplishing the same task on macOS is harder: steps 1, 2, 4, and 5 are the same, but step 3 is missing the convenience functions from iOS. Yes, you can create an **NSImage** in a similar way to a **UIImage** (macOS even has a URL-based initializer, which is a bonus!), but getting that image into PNG data is non-trivial.

There are several approaches you could take, but there's only one I consider even vaguely straightforward: get the TIFF representation of the image (this is a property on **NSImage**), create a bitmap representation from that, then convert that to a PNG for writing.

Here's the code:

```
func challenge35(in directory: String) {
    let fm = FileManager.default
    let directoryURL = URL(fileURLWithPath: directory)
    guard let files = try? fm.contentsOfDirectory(at:
        directoryURL, includingPropertiesForKeys: nil) else { return }

    for file in files {
        guard file.pathExtension == "jpeg" || file.pathExtension
        == "jpg" else { continue }
        guard let image = NSImage(contentsOf: file) else
        { continue }
        guard let tiffData = image.tiffRepresentation else
        { continue }
        guard let imageRep = NSBitmapImageRep(data: tiffData)
        else { continue }
        guard let png = imageRep.representation(using: .png,
        properties: [:]) else { continue }

        let newfilename =
        file.deletingPathExtension().appendingPathExtension("png")
```

## Files

```
    _ = try? png.write(to: newFilename)
}
}
```

Yes, that's a lot of **guard** in one function, but it does help avoid a mess of optionality!

# Challenge 36: Print error lines

**Difficulty:** Taxing

Write a function that accepts a path to a log file on disk, and returns how many lines start with “[ERROR]”. The log file could be as large as 10GB, but may also not exist.

## Sample input and output

- If the file contains 100,000,000 lines and eight start with “[ERROR]” your function should return 8.
- If the file does not exist, cannot be loaded, or contains zero lines starting with “[ERROR]” your function should return 0.

## Hints

**Hint #1:** You can use the `contentsOfFile` initializer for strings, but only if you want to be laughed out of the room.

**Hint #2:** This is a job for `FileHandle`, which opens a file and reads chunks of a size you specify using `readData(ofLength:)`.

**Hint #3:** `FileHandle` doesn’t care about line breaks, so it will read as much data as you ask. It’s down to you to use `range(of:)` to find zero or more line breaks inside the chunk you read.

**Hint #4:** To make things neater, I created a custom class to store my file reading code – you might want to consider doing the same.

## Solution

This is a challenge that ought to be easy, but sadly is far harder than you think because Swift doesn’t have any neat APIs for streaming data. A number of brave people have stepped up to

## Files

try to create third-party components, not least Dave DeLong with his Objective-C **DD.FileReader** component, but clearly it wouldn't be a great answer to this question if you just pulled in a third-party component!

So, instead I want to show you a solution that is a lot simpler. I took the structure of Dave's **DD.FileReader**, converted it to Swift, stripped out everything that wasn't essential, then refactored what remained to make it as simple as possible. I'm afraid even then it's not simple enough to remember, but hopefully you can at least see the *structure* of the solution and that might be enough to help you recreate it if needed.

Here's how the code works:

1. It's a new class called **ChunkedFileReader** that is responsible for opening a file and reading lines until the end of the file is reached.
2. I've made it read lines 4 characters at a time (**chunkSize**), which is an absurdly small amount in production, but great as a test so you can step through it at runtime to see how it works. (A chunk size of between 1024 and 8192 bytes would be more sensible for production.)
3. You initialize the class by passing it a filename. It then creates a buffer, which is what it reads the file into over time.
4. Every time you call its **readLine()** method, it attempts to find the next line break. It does this by reading **chunkSize** bytes into its buffer and scanning for the delimiter "\n".
5. If no line break is found, it keeps reading chunks and placing them into the buffer.
6. Eventually, either a line break is found or the file ends. Either way, if the buffer has data inside it gets converted into a string and sent back.

I used "\n" for the delimiter – the marker for where lines end – which suits this challenge nicely.

Here's my **ChunkedFileReader** class, with inline comments:

```
class ChunkedFileReader {  
    var fileHandle: FileHandle?
```

```
// we keep adding to the buffer until eventually we find a
line break or hit the end of the file
var buffer: Data

// this should be between 1024 and 8192 for production; it
determines how much of the file is read in each step
let chunkSize: Int = 4

// this determines what we consider to be an end of line
let delimiter = "\n".data(using: .utf8)!

init(path: String) {
    // open the file handle and prepare the buffer
    fileHandle = FileHandle(forReadingAtPath: path)
    buffer = Data(capacity: chunkSize)
}

func readLine() -> String? {
    // find the delimiter in our buffer
    var rangeOfDelimiter = buffer.range(of: delimiter)

    // loop until we finally find a delimiter
    while rangeOfDelimiter == nil {
        // try to read a chunk from the file handle, or bail
        // out if it didn't work
        guard let chunk = fileHandle?.readData(ofLength:
chunkSize) else { return nil }

        if chunk.count == 0 {
            // the chunk was read successfully, but was empty –
            // we reached the end of the file
            return nil
        }

        rangeOfDelimiter = chunk.range(of: delimiter)
        buffer.append(chunk)
    }
}
```

## Files

```
        if buffer.count > 0 {
            // the buffer has something left in it; send it
            back, and make sure to clear the buffer's remnants when it's
            finished
            defer { buffer.count = 0 }
            return String(data: buffer, encoding: .utf8)
        }

        // we reached the end of the file and the buffer
        was empty; send back nil
        return nil
    } else {
        // we read some data; append it to our buffer
        buffer.append(chunk)

        // now re-scan for the delimiter
        rangeOfDelimiter = buffer.range(of: delimiter)
    }
}

// we can only make it here if we found a delimiter, but
it might be anywhere inside our buffer; we want to pull out
everything in our buffer from the start up to where the
delimiter lies
let rangeOfLine = Range(0 ..<
rangeOfDelimiter!.upperBound)

// convert that range of our buffer into a string
let line = String(data: buffer.subdata(in: rangeOfLine),
encoding: .utf8)

// then remove it from the buffer
```

## Challenge 36: Print error lines

```
        buffer.removeSubrange(rangeOfLine)

        // send the line back, removing the trailing line break
at the end.

        line?.trimmingCharacters(in: .whitespacesAndNewlines)
    }
}
```

So: it reads in chunk after chunk from the file until it eventually finds a line break, at which point it pulls out everything from the start of the buffer up to the line break and returns it as a line. The remainder of the buffer stays in place, waiting for the next **readLine()** call.

Of course, all that doesn't actually satisfy the challenge that was set, it just puts the foundations in place. And in fact it does nearly all the work, so all that remains is to create a new **ChunkedFileReader** instance and continually call its **readLine()** method and look for the string prefix “[ERROR]”:

```
func challenge36(filename: String) -> Int {
    var totalErrors = 0

    let reader = ChunkedFileReader(path: filename)

    while let line = reader.readLine() {
        if line.hasPrefix("[ERROR]") {
            totalErrors += 1
        }
    }

    return totalErrors
}
```

Done!

## Files

It's a real shame that this task – a fairly fundamental one, if you ask me – takes so much work. Perhaps Apple will add their own chunked file reader to the standard library in the future...

# **Chapter 4**

## Collections

# Challenge 37: Count the numbers

**Difficulty:** Easy

Write an extension for collections of integers that returns the number of times a specific digit appears in any of its numbers.

## Sample input and output

- The code `[5, 15, 55, 515].challenge37(count: "5")` should return 6.
- The code `[5, 15, 55, 515].challenge37(count: "1")` should return 2.
- The code `[55555].challenge37(count: "5")` should return 5.
- The code `[55555].challenge37(count: "1")` should return 0.

## Hints

**Hint #1:** You'll need to extend the **Collection** type with a specific constraint rather than a protocol constraint.

**Hint #2:** If you convert each number to a string, you can loop over its characters.

**Hint #3:** If you were functionally inclined, you could solve this challenge using `reduce()` and `filter()`.

## Solution

To count the number of a specific digit inside an integer, the easiest thing to do is convert it to a string then check each character. Swift will only let you do this if you specify the correct constraint, which in this challenge ought to be **Iterator.Element == Int**, like this:

```
extension Collection where Iterator.Element == Int {  
    func challenge37a(count: Character) -> Int {
```

## Challenge 37: Count the numbers

```
var total = 0

// loop over every element
for item in self {
    // stringify this integer
    let str = String(item)

    // loop over letter in the string
    for letter in str {
        if letter == count {
            total += 1
        }
    }
}

return total
}
```

A more interesting solution is to use `reduce()` to whittle down the array of integers into a single number, then use `filter()` on each item to pick out characters that match the input digit:

```
func challenge37b(count: Character) -> Int {
    return self.reduce(0) {
        $0 + String($1).filter { (char: Character) -> Bool in
            char == count }.count
    }
}
```

That's certainly much shorter, and arguably much clearer in its intent, but it's unlikely to perform faster.

# Challenge 38: Find N smallest

**Difficulty:** Easy

Write an extension for all collections that returns the  $N$  smallest elements as an array, sorted smallest first, where  $N$  is an integer parameter.

## Sample input and output

- The code `[1, 2, 3, 4].challenge38(count: 3)` should return `[1, 2, 3]`.
- The code `["q", "f", "k"].challenge38(count: 10)` should return `["f", "k", "q"]`.
- The code `[256, 16].challenge38(count: 3)` should return `[16, 256]`.
- The code `[String]().challenge38(count: 3)` should return an empty array.

## Hints

**Hint #1:** You'll need to extend the **Collection** type with a constraint.

**Hint #2:** Finding the smallest of any value requires using the `<` operator, which is guaranteed to exist when something conforms to **Comparable**.

**Hint #3:** The collection might contain fewer than  $N$  items.

**Hint #4:** The solution is made more interesting by the requirement to return a variable number of results.

**Hint #5:** If you want to avoid complexity, use `sorted()`.

## Solution

Finding the single smallest value in a collection is easy enough; finding the two smallest values in a collection is fractionally harder, because you need to move the smallest to smaller

## Challenge 38: Find N smallest

if a suitable small new value comes in.

Finding the  $N$  smallest numbers is more difficult again, or would be if you followed the same technique. Rather than keeping individual variables for smallest and smaller, you would need to keep a sorted array of small values, inserting new values in the correct position as they came in. This isn't hard to code, but neither is it terribly fast, and it becomes pretty disastrously slow if you had to insert lots of numbers.

A solution that's significantly simpler and probably significantly faster too is to use the built-in **sorted()** method followed by **prefix()**, which returns the first  $N$  items in a collection. If there are fewer items than you request, **prefix()** will return as many as it can – perfect for this challenge.

So, the smart solution is as follows:

```
extension Collection where Iterator.Element: Comparable {
    func challenge38(count: Int) -> [Iterator.Element] {
        let sorted = self.sorted()
        return Array(sorted.prefix(count))
    }
}
```

# Challenge 39: Sort a string array by length

**Difficulty:** Easy

Extend collections with a function that returns an array of strings sorted by their lengths, longest first.

## Sample input and output

- The code `["a", "abc", "ab"].challenge39()` should return `["abc", "ab", "a"]`.
- The code `["paul", "taylor", "adele"].challenge39()` should return `["taylor", "adele", "paul"]`.
- The code `[String]().challenge39()` should return `[]`.

## Hints

**Hint #1:** You'll need to extend the **Collection** type with a specific constraint rather than a protocol constraint.

**Hint #2:** You should use the built-in **sorted()** method.

**Hint #3:** You can provide a custom closure to **sorted()** to affect how it works.

## Solution

This is a simple test that requires you to understand protocol extensions as well as using closures with built-in methods. There's also some scope to discuss reverse sorting – there *is* a built-in **reversed()** method, but if you're writing a custom closure anyway you might as well just flip the operator from `<` to `>`.

## Challenge 39: Sort a string array by length

All in, it takes just one line of code to solve, as long as you embed that line inside the right method and extension definitions:

```
extension Collection where Iterator.Element == String {  
    func challenge39() -> [String] {  
        return self.sorted { $0.count > $1.count }  
    }  
}
```

# Challenge 40: Missing numbers in array

**Difficulty:** Easy

Create a function that accepts an array of unsorted numbers from 1 to 100 where zero or more numbers might be missing, and returns an array of the missing numbers.

## Sample input and output

If your test array were created like this:

```
var testArray = Array(1...100)
testArray.remove(at: 25)
testArray.remove(at: 20)
testArray.remove(at: 6)
```

Then your method should **[7, 21, 26]**, because those are the numbers missing from the array.

## Hints

**Hint #1:** There's a naïve solution involving arrays, but it's very slow.

**Hint #2:** You should try using a **Set**, which has a significantly faster **contains()** method.

**Hint #3:** You can compute the different between two sets using **symmetricDifference()**.

## Solution

If you create a new array with all numbers present, then you can loop over each number and check whether it exists in the input array. Any missing numbers get added to an array, and that

## Challenge 40: Missing numbers in array

array gets sent back as the return value for the method – simple, right?

Here it is in Swift:

```
func challenge40a(input: [Int]) -> [Int] {
    let correctArray = Array(1...100)
    var missingNumbers = [Int]()

    for number in correctArray {
        if !input.contains(number) {
            missingNumbers.append(number)
        }
    }

    return missingNumbers
}
```

The problem is, that solution is slow. We know that the input array will contain 100 items give or take some number of missing items, which means **contains()** will have to search through as many as 100 items before it can confirm it has found a number.

A much faster solution is to use a set, which has an O(1) **contains()** method – it runs at the same speed no matter whether it contains one item or one thousand. You can construct a set directly from the input array, and immediately the code will run around 10x faster for our test case:

```
func challenge40b(input: [Int]) -> [Int] {
    let correctArray = Array(1...100)
    let inputSet = Set(input)
    var missingNumbers = [Int]()

    for number in correctArray {
        if !inputSet.contains(number) {
```

## Collections

```
    missingNumbers.append(number)
}
}

return missingNumbers
}
```

A third possible option is to use set algebra: if we construct a set from both the correct answer and from the input array, we can calculate the missing numbers by subtracting one from the other, then sorting the result.

This involves a little more work: this creates a new set with only elements that appear in both, creates an array out of that set, then sorts it. Still, it's at least worth reading the code:

```
func challenge40c(input: [Int]) -> [Int] {
    let inputSet = Set(input)
    let testSet = Set(1...100)
    return Array(testSet.subtracting(inputSet)).sorted()
}
```

# Challenge 41: Find the median

**Difficulty:** Easy

Write an extension to collections that accepts an array of **Int** and returns the median average, or nil if there are no values.

**Tip:** The mean average is the sum of some numbers divided by how many there are. The *median* average is the middle of a sorted list. If there is no single middle value – e.g. if there are eight numbers - then the median is the mean of the two middle values.

## Sample input and output

- The code `[1, 2, 3].challenge41()` should return 2.
- The code `[1, 2, 9].challenge41()` should return 2.
- The code `[1, 3, 5, 7, 9].challenge41()` should return 5.
- The code `[1, 2, 3, 4].challenge41()` should return 2.5.
- The code `[Int]().challenge41()` should return nil.

## Hints

**Hint #1:** You'll need to extend **Collection** with a specific constraint.

**Hint #2:** The method should return **Double?** because it might be a whole number, it might be a mean average of two numbers, or it might be nil if the collection is empty.

**Hint #3:** If you divide an odd integer by two, Swift will round down.

**Hint #4:** If you divide an even-numbered collection's count by two, you'll get the highest of the two values you need for your mean.

**Hint #5:** Make life easy for yourself: sort the collection first.

## Solution

Everyone knows how to do a mean average, because that's by far the most common. The *median* average is a little bit out of the ordinary, so it might cause you to think back to school just a little.

As a reminder, the median average of an array is the number that lies in its center once the array is sorted. If you have two items in the center – i.e., you have an array with an even number of items – then the median is the mean of the two center items.

So, this is a fairly elementary challenge, spiced up a little by the need for you to frame it as an extension. Hopefully you figured out the extension constraint should be **Iterator.Element == Int**, at which point the remainder is plain sailing:

```
extension Collection where Iterator.Element == Int {
    func challenge41() -> Double? {
        guard count != 0 else { return nil }

        // sort the items so we can find the center point
        let sorted = self.sorted()
        let middle = sorted.count / 2

        if sorted.count % 2 == 0 {
            // return mean average of two center items
            return Double(sorted[middle] + sorted[middle - 1]) / 2
        } else {
            // return the single center item
            return Double(sorted[middle])
        }
    }
}
```

## Challenge 41: Find the median

Not too hard, I think. Perhaps now you could try implementing a mode average too?

# Challenge 42: Recreate firstIndex(of:)

**Difficulty:** Easy

Write an extension for all collections that reimplements the **firstIndex(of:)** method.

**Tip:** This is one of the easiest standard library methods to reimplement, so please give it an especially good try before reading any hints.

## Sample input and output

- The code `[1, 2, 3].challenge42(firstIndexOf: 1)` should return 0.
- The code `[1, 2, 3].challenge42(firstIndexOf: 3)` should return 2.
- The code `[1, 2, 3].challenge42(firstIndexOf: 5)` should return nil.

## Hints

**Hint #1:** You will need to extend **Collection** using a constraint on the type of element it stores.

**Hint #2:** Your return type should be **Int?** because the item might not exist in the collection.

**Hint #3:** This would be a good time to use **enumerated()** to retrieve items and their index from a collection.

## Solution

This is a marvelously simple challenge that is perfect for helping someone learn algorithms – it's simple enough you can write it down on a piece of paper first time, but also demonstrates the importance of optional and protocol extensions all in one.

## Challenge 42: Recreate firstIndex(of:)

To solve the challenge, you need to start by extending the **Collection** protocol using a constraint that elements must conform to **Equatable**. Without that constraint you can't use `==`, and therefore can't tell whether you've found the correct item in the collection.

Once that's done, you just need to loop through all the items in your collection, perhaps using `enumerated()`, then return the matching index.

Here's an example solution:

```
extension Collection where Iterator.Element: Equatable {
    func challenge42(firstIndexOf search: Iterator.Element) -> Int? {
        for (index, item) in self.enumerated() {
            if item == search {
                return index
            }
        }

        return nil
    }
}
```

# Challenge 43: Linked lists

**Difficulty:** Easy

Create a linked list of lowercase English alphabet letters, along with a method that traverses all nodes and prints their letters on a single line separated by spaces.

**Tip #1:** This is several problems in one. First, create a linked list data structure, which itself is really two things. Second, use your linked list to create the alphabet. Third, write a method that traverses all nodes and prints their letters.

**Tip #2:** You should use a class for this. Yes, really.

**Tip #3:** Once you complete your solution, keep a copy of the code – you'll need it for future challenges!

## Sample input and output

- The output of your code should be the English alphabet printed to the screen, i.e. “a b c d ... x y z”.

## Hints

**Hint #1:** If your first instinct was to create your new data types as a struct, it shows you're a sound Swift developer. Sadly, I'm afraid that approach won't work here because structs can't have stored properties that reference themselves.

**Hint #2:** Your second instinct might be to use an enum. This makes creation tricksy because you would need to change the associated value after creating the enum.

**Hint #3:** Even though this challenge uses alphabet letters, aim to make your class generic – it shows smart forward thinking, and is only fractionally harder than using a specific data type.

**Hint #4:** There are lots of hacky ways to loop over the alphabet. The only sensible way is to use "**abcdefghijklmnopqrstuvwxyz**" – it's not hard to write, is self-documenting, and quite safe.

**Hint #5:** You should create two data types: one for a node, which contains its character and link to the next node in the list, and one for the overall linked list, which contains a property for the first node in the list as well as the print method.

**Hint #6:** The `print()` function has a **terminator** parameter.

## Solution

Linked lists are a simple data structure that you ought to be able to code from scratch in a couple of minutes. At first it's easy to see them as hobbled arrays, but the truth is they have their own advantages and disadvantages just like any data structure.

For example, if you want to insert a new value into the middle of an array, the `insert()` method is  $O(n)$  – it takes longer and longer depending on the size of the array. This is because Swift has to move other items down one space before inserting the new item. In comparison, inserting something into the middle of a linked list is  $O(1)$  because you just change two pointers. On the other hand, jumping to an arbitrary position in an array is  $O(1)$ , whereas it's  $O(n)$  in a linked list because you need to start at the first node and work your way down the chain.

Let's jump in to the solution, starting with a custom class to handle nodes:

```
class LinkedListNode<T> {
    var value: T
    var next: LinkedListNode?

    init(value: T) {
        self.value = value
    }
}
```

## Collections

```
}
```

So, each node has a value, as well as an optional link to its successor. In the initializer we only set the value, because we'll be setting the **next** property after creation.

Wrapping that node type in a parent class is trivial:

```
class LinkedList<T> {
    var start: LinkedListNode<T>?
}
```

We'll come back to that soon to add a **printNodes()** method.

In order to create our linked list, we're going to start by creating two variables:

```
var list = LinkedList<Character>()
var previousNode: LinkedListNode<Character>? = nil
```

The **previousNode** will be used to create our chain – each time we get a letter to create, we'll see the **next** property of **previousNode** to the most recently created node.

This is all done using a loop. As I said in the hints, the smartest way to loop over the letters of the alphabet is just to type them in a string and treat it like an array.

Inside the loop we'll create a new **LinkedListNode** for the current letter. If **previousNode** is **nil** it means this is the first letter to be created, so we'll set **list.start** to be the new node. Otherwise we have a predecessor – the node that should link to this new node – so we'll update its **next** property to point at the new node.

Here's all that in code:

```
for letter in "abcdefghijklmnopqrstuvwxyz" {
    let node = LinkedListNode(value: letter)
```

```

if let predecessor = previousNode {
    predecessor.next = node
} else {
    list.start = node
}

previousNode = node
}

```

So, at this point we've accomplished two of the three parts to this challenge: we've created our own linked list data type, and created a list out of the alphabet. All that remains is traversing the list to print out all its values on a single line, separated by spaces.

Once you know that the **print()** function has a **terminator** parameter that lets you specify something other than a line break to use after the line is printed, this method is pretty straightforward:

1. Pick out the first node in the list using the **start** property and make that the current node.
2. While the current node is not nil, print out its value with a space for the terminator.
3. Update the current node to be equal to the value of its **next** property, thereby moving to the next element in the list.

That's it! Here's the code:

```

func printNodes() {
    var currentNode = start

    while let node = currentNode {
        print(node.value, terminator: " ")
        currentNode = node.next
    }
}

```

## Collections

For reference, here's my complete solution to this challenge:

```
class LinkedListNode<T> {
    var value: T
    var next: LinkedListNode?

    init(value: T) {
        self.value = value
    }
}

class LinkedList<T> {
    var start: LinkedListNode<T>?

    func printNodes() {
        var currentNode = start

        while let node = currentNode {
            print(node.value, terminator: " ")
            currentNode = node.next
        }
    }
}

var list = LinkedList<Character>()
var previousNode: LinkedListNode<Character>? = nil

for letter in "abcdefghijklmnopqrstuvwxyz" {
    let node = LinkedListNode(value: letter)

    if let predecessor = previousNode {
        predecessor.next = node
    } else {
```

## Challenge 43: Linked lists

```
list.start = node  
}  
  
previousNode = node  
}  
  
list.printNodes()
```

# Challenge 44: Linked list mid-point

**Difficulty:** Easy

Extend your linked list class with a new method that returns the node at the mid point of the linked list using no more than one loop.

**Tip:** If the linked list contains an even number of items, returning the one before or the one after the center is acceptable.

## Sample input and output

- If the linked list contains 1, 2, 3, 4, 5, your method should return 3.
- If the linked list contains 1, 2, 3, 4, your method may return 2 or 3.
- If the linked list contains the English alphabet, your method may return M or N.

## Hints

**Hint #1:** It's easy to solve this in two passes, but only fractionally harder to solve it in one.

**Hint #2:** If you use fast enumeration – `for i in items` – you move over one item at a time. Can you think of a way of moving over more than one item?

**Hint #3:** Once you pull out two items at the same time, you can make them move at different speeds through the list.

**Hint #4:** If you move pointer A through the list one item at a time, and pointer B through the list two items at a time, by the time pointer B reaches the end where will pointer A be?

## Solution

Moving through collections at different speeds is a useful way to solve many problems, including this one and another coming soon.

In this case, the clue is in the challenge description: we aren't allowed to use more than one loop, so we have no way of knowing where the middle of the loop is unless we look ahead. Fortunately, the end of the linked list is more or less twice as far as the half-way point – that's obvious when you think about it, although it *is* more or less because even-numbered lists don't have a precise center.

So, to find the center of a linked list we need to create two pointers: a slow-moving one and a fast-moving one. We can then start our loop: for as long as the fast pointer is not nil, and its **next** node is also not nil, we'll move it forward two places. At the same time, we'll move the slow pointer forward one. When the loop finishes – i.e., when either the fast pointer is nil or its next value is nil – we've reached the end, at which point the slow pointer contains our center point.

Here's my solution:

```
var centerNode: LinkedListNode<T>? {
    var slow = start
    var fast = start

    while fast != nil && fast?.next != nil {
        slow = slow?.next
        fast = fast?.next?.next
    }

    return slow
}
```

# Challenge 45: Traversing the tree

**Difficulty:** Easy

**Note:** this challenge cannot be attempted until you have first completed challenge 54.

Write a new method for your binary search tree that traverses the tree in order, running a closure on each node.

**Tip:** Traversing a node in order means visiting its left value, then visiting its own value, then visiting its right value.

## Sample input and output

Assuming a binary tree created from the array [2, 1, 3]:

- The code `tree.root?.traverse { print($0.key) }` should print 1, 2, 3.
- The code `var sum = 0; tree.root?.traverse { sum += $0.key }; print(sum)` should print 6.
- The code `var values = [Int](); tree.root?.traverse { values.append($0.key) }; print(values.count)` should print 3.

## Hints

**Hint #1:** Your entire function can be just three lines of code. Yes, it really is that easy – hurray for recursion!

**Hint #2:** You can write this method for the binary tree class or for its nodes; it really doesn't matter. I chose to write it for the nodes so that I can print partial trees.

**Hint #3:** Make sure it accepts a closure parameter that itself accepts one parameter (your `Node<T>` equivalent) and returns void.

**Hint #4:** Remember the left and/or right node may not exist.

## Solution

Once you've expended all the effort creating a binary tree, working with it is remarkably easy and this challenge is a great example of that.

Traversing a binary tree in order means going through each node, applying a closure to its left node, applying a closure to itself, and applying a closure to its right node. There are an unknown number of nodes in any position in the binary tree, so this is a perfect candidate for recursion.

Consider a tree with one item at the top, two items beneath, four items beneath that, and eight items at the bottom. Starting from the top, we need to move its left node, then the left node again, then the left node again, then the last left node. That's our first value to print. We can then move up one level to print the parent value, then move down again to print the right node. Next we move up to the grandparent and print its value, then its right node, then move up again, and so on.

This might sound hard in theory, but in code it's almost magical in its simplicity. Here's my solution in full:

```
func traverse(_ body: (Node<T>) -> Void) {
    left?.traverse(body)
    body(self)
    right?.traverse(body)
}
```

*Boom.* Another challenge aced!

# Challenge 46: Recreate map()

**Difficulty:** Tricky

Write an extension for all collections that reimplements the **map()** method.

## Sample input and output

- The code `[1, 2, 3].challenge46 { String($0) }` should return `["1", "2", "3"]`
- The code `["1", "2", "3"].challenge46 { Int($0)! }` should return `[1, 2, 3]`.

## Hints

**Hint #1:** You'll need to extend the **Collection** type.

**Hint #2:** Your transformation function should accept a parameter of type **Iterator.Element**, but must return a generic parameter.

**Hint #3:** You should accept transformation functions that throw, but you don't want to handle any exceptions in your mapping method.

**Hint #4:** Non-throwing functions are sub-types of throwing functions.

**Hint #5:** You really ought to use **rethrows** to avoid irritating users who use non-throwing functions.

## Solution

This is an easy problem to solve if you're an experienced Swift developer, but tricky or perhaps even taxing if you've only touched small amounts of the language – anyone who has read Pro Swift ought to have found this a walk in the park!

To make a **map()** method truly useful, you need to draw on several of Swift's power features:

- Generics, because you don't know what kind of data type will be returned. In the example input and output I specifically make the return types be different to the input types.
- Closures, because that's how your transformation function will be specified.
- Throwing and rethrowing functions. The former because the transformation function ought to be able to throw; the latter because you don't want to handle exceptions inside your mapping method, but also don't want to mark the whole method as throwing because that would be annoying at the call site.
- Protocol extensions, because you're required to extend all collections, not just arrays or dictionaries.

The truly incredible thing is how little code it takes to do all that.

```
// extend all collections
extension Collection {
    // add a generic method that accepts a closure operating on
    // our element type and returns a new type, with the whole method
    // returning an array of that type
    func challenge46<T>(_ transform: (Iterator.Element) throws -> T) rethrows -> [T] {
        // create the return array
        var returnValue = [T]()
        
        // loop over all our items, trying the transformation and
        // appending it to our return
        for item in self {
            returnValue.append(try transform(item))
        }
        
        // send back the return value
        return returnValue
    }
}
```

## Collections

}

The only thing that might have thrown you there is the use of “throws” and “rethrows”. This is covered in detail in Pro Swift, but the TL;DR version is this: marking the parameter with **throws** means only that it *might* throw, not that it *will* throw, and marking the whole thing as **rethrows** means it need be used with **try/catch** only when its parameter really does throw.

# Challenge 47: Recreate min()

**Difficulty:** Tricky

Write an extension for all collections that reimplements the **min()** method.

## Sample input and output

- The code `[1, 2, 3].challenge47()` should return 1.
- The code `["q", "f", "k"].challenge47()` should return “f”.
- The code `[4096, 256, 16].challenge47()` should return 16.
- The code `[String]().challenge47()` should return nil.

## Hints

**Hint #1:** You’ll need to extend the **Collection** type with a constraint.

**Hint #2:** Finding the smallest of any value requires using the `<` operator, which is guaranteed to exist when something conforms to **Comparable**.

**Hint #3:** The collection might be empty, so you’ll need to return an optional value.

**Hint #4:** You can’t compare an optional value against a non-optional one

**Hint #5:** You can solve this quite beautifully using **reduce()**.

## Solution

As you might expect, there’s a naïve solution, a smart solution, and a sneaky solution. I want to show you all three, because there’s a good chance you’ll learn something new along the way – you’re getting good value for money, right?

## Collections

Let's take a look at the naïve solution first:

```
extension Collection where Iterator.Element: Comparable {
    func challenge47a() -> Iterator.Element? {
        var lowest: Iterator.Element?

        for item in self {
            if let unwrappedLowest = lowest {
                if item < unwrappedLowest {
                    lowest = item
                }
            } else {
                lowest = item
            }
        }

        return lowest
    }
}
```

So, make the lowest value an optional of the same type as our element, then loop through all items. For each item, unwrap our lowest and check whether the new item is lower; if so, replace it and move on.

That certainly works, but the optionality here is just getting in the way. A smarter solution is to jettison optionality entirely, because if we can't get at least one value from the collection we might as well early return.

So, we can use **guard** to pull out **self.first**. If that returns nil it means the collection is empty so we can return nil immediately. However, if it *isn't* empty it means we definitely have a return value and can therefore forget about optionality the rest of the time.

Here's how this improved solution works:

## Challenge 47: Recreate min()

```
func challenge47b() -> Iterator.Element? {
    guard var lowest = self.first else { return nil }

    for item in self {
        if item < lowest {
            lowest = item
        }
    }

    return lowest
}
```

That's significantly shorter and easier to read. Optionality is a firm guarantee of program safety, but it's also an epic pain in the backside – if you can ditch it, you should do so.

The current code has a small problem, which is that the first item in the collection is read twice: once for the initial value, and once again when the loop starts. Honestly, that cost is tiny, and of course becomes increasingly small as the size of the collection increases.

However, it's possible that your collection stores custom objects that are expensive to compare, so I want to show you how to fix this problem.

All collections are backed by an iterator, which is what produces elements and allows us to constrain extensions using **Iterator.Element**. They also come with a **makeIterator()** method, that allows you to move through the entire collection and pull out items by calling its **next()** method. The first time you call **next()** you'll receive the first item; the second time you call it you'll get the second, and so on. Helpfully, you can call these methods whenever and wherever you want – as long as you use the same iterator, your position will be preserved.

So, rather than using **self.first** for our **guard** statement, we'll instead create an iterator then use the first return value from its **next()** method. This means we can continue to loop over the iterator using a **while** loop and **next()**, and won't compare the first value to itself.

So, here's the doubly improved code:

## Collections

```
func challenge47c() -> Iterator.Element? {
    var it = makeIterator()
    guard var lowest = it.next() else { return nil }

    while let item = it.next() {
        if item < lowest {
            lowest = item
        }
    }

    return lowest
}
```

OK, enough of that solution. Let's take a look at some alternatives, starting with **reduce()**. You can solve this challenge in just two lines of code:

```
func challenge47d() -> Iterator.Element? {
    guard let lowest = self.first else { return nil }
    return reduce(lowest) { $1 < $0 ? $1 : $0 }
}
```

That probably isn't the fastest, but you must admit it's beautiful in its simplicity. Using **reduce()** with the ternary operator effectively makes this into a fight to the death: the winner of the previous round advances to the next one, until the method finally returns.

That code does suffer from the same problem we already tackled: it evaluates the first element twice. Sadly this is difficult to avoid with **reduce()**, because you can't reduce an iterator. However, Swift has a workaround that might interest you – it's not especially efficient if you want to squeeze out maximum performance, but it does let you treat an iterator like any other sequence.

The workaround is the **IteratorSequence** data type. It is initialized from an iterator, but includes all the functionality you're used to in sequences, such as **for-in** loops – and, critically

## Challenge 47: Recreate min()

here, **reduce()**. So, we could rewrite the previous solution using **IteratorSequence** like so:

```
func challenge47e() -> Iterator.Element? {
    var it = makeIterator()
    guard let lowest = it.next() else { return nil }
    return IteratorSequence(it).reduce(lowest) { $1 < $0 ? $1 :
$0 }
}
```

I should add that there's a cost to converting an iterator to a sequence, and it's probably greater than the cost of evaluating one item twice.

Finally, a sneaky solution: call **sorted()** then return its **first** property. This ensures all items are placed in their correct order, so **first** is guaranteed to be the smallest item, or nil if the collection is empty. Here it is in code:

```
func challenge47f() -> Iterator.Element? {
    return self.sorted().first
}
```

This solution certainly works, and involves writing almost no code, but it's far from the most efficient. Keep in mind that **sorted()** has to do multiple comparisons in order to place items in their correct order, as well as multiple moves, so it involves far more operations than are required to solve this challenge.

# Challenge 48: Implement a deque data structure

**Difficulty:** Tricky

Create a new data type that models a double-ended queue using generics, or *deque*. You should be able to push items to the front or back, pop them from the front or back, and get the number of items.

**Tip:** It's pronounced like "deck".

## Sample input and output

Once your data structure has been created, this code should compile and run cleanly:

```
var numbers = deque<Int>()
numbers.pushBack(5)
numbers.pushBack(8)
numbers.pushBack(3)
assert(numbers.count == 3)
assert(numbers.popFront()! == 5)
assert(numbers.popBack()! == 3)
assert(numbers.popFront()! == 8)
assert(numbers.popBack() == nil)
```

## Hints

**Hint #1:** Use an internal array for your data.

**Hint #2:** If you used a class for this, expect to be questioned carefully as to why you didn't choose a struct.

## Challenge 48: Implement a deque data structure

**Hint #3:** You'll need to declare your whole data type as being generic, e.g. **struct deque<T>** {.

**Hint #4:** The **popBack()** and **popFront()** method should return optionals, because the deque might be empty.

**Hint #5:** You'll need to mark your methods as **mutating**.

**Hint #6:** Make sure **count** is a property rather than a method. Something like **var count: Int { return array.count }** ought to do it.

## Solution

I graded this one tricky, but really it's only tricky if you rarely use generics – as data structures go, deques are one of the easiest and so this should have caused you no trouble as long as you are comfortable with generics.

With my specific test case code above, you should have written something almost identical to my solution below:

```
struct deque<T> {
    private var array = [T]()

    var count: Int {
        return array.count
    }

    mutating func pushBack(_ obj: T) {
        array.append(obj)
    }

    mutating func pushFront(_ obj: T) {
        array.insert(obj, at: 0)
    }
}
```

## Collections

```
}

mutating func popBack() -> T? {
    return array.popLast()
}

mutating func popFront() -> T? {
    if array.isEmpty {
        return nil
    } else {
        return array.removeFirst()
    }
}
```

The only real piece of interest there is the **popFront()** method, because Swift arrays don't have a **popFirst()** method we can map to.

# Challenge 49: Sum the even repeats

**Difficulty:** Tricky

Write a function that accepts a variadic array of integers and return the sum of all numbers that appear an even number of times.

## Sample input and output

- The code `challenge49(1, 2, 2, 3, 3, 4)` should return 5, because the numbers 2 and 3 appear twice each.
- The code `challenge49(5, 5, 5, 12, 12)` should return 12, because that's the only number that appears an even number of times.
- The code `challenge49(1, 1, 2, 2, 3, 3, 4, 4)` should return 10.

## Hints

**Hint #1:** This is a perfect use for `NSCountedSet`.

**Hint #2:** But: `NSCountedSet` doesn't use generics, so you'll need to typecast somehow. Expect to be judged on your method of typecasting!

**Hint #3:** You'll need to use modulus to find numbers that are repeated an even number of times.

**Hint #4:** You'll need to declare your parameter as `numbers: Int...`

## Solution

This is really three problems in one: count items in an array, select specific counts, and sum

## Collections

them. None of those are hard by themselves, but combined I think the whole challenge warrants a tricky grade because it can be hard to know where to start.

Fortunately, we have **NSCountedSet**, my personal favorite Foundation data type. This will do the hard work counting items in an array, so all we have to do is run through all its results to find those that have even counts.

One fly in the ointment is that **NSCountedSet** doesn't support generics, so you'll need to either typecast the whole thing or individual elements as you need them.

Given that we know we're creating the **NSCountedSet** from an array of integers, we could create an integer array like this:

```
let array = countedSet.allObjects as! [Int]
```

Alternatively, we could avoid creating a new array and instead just typecast each item as we go:

```
for case let item as Int in countedSet {  
    ...  
}
```

It's tempting to imagine the former is faster, but it's rarely a good idea to create new variables if they aren't really needed. In this case the cost of bridging between **Any** and **Int** is likely to be small, and taking the **for case let** approach has the benefit of removing the ugly **as!** typecast.

Here are both solutions in full, starting with typecasting each item in the loop:

```
func challenge49a(numbers: Int...) -> Int {  
    let counted = NSCountedSet(array: numbers)  
    var sum = 0  
  
    for case let item as Int in counted {  
        ...  
    }  
    ...  
}
```

## Challenge 49: Sum the even repeats

```
    if counted.count(for: item) % 2 == 0 {
        sum += item
    }
}

return sum
}
```

And here's the slower solution that converts the counted set into an integer array before the loop:

```
func challenge49b(numbers: Int...) -> Int {
    let counted = NSCountedSet(array: numbers)
    let array = counted.allObjects as! [Int]
    var sum = 0

    for item in array {
        if counted.count(for: item) % 2 == 0 {
            sum += item
        }
    }

    return sum
}
```

Getting rid of an unnecessary **as!** typecast always feels good, but in this case the first solution also benefits from using the **for case let** syntax – one of Swift's real gems.

# Challenge 50: Count the largest range

**Difficulty:** Tricky

Write a function that accepts an array of positive and negative numbers and returns a closed range containing the position of the contiguous positive numbers that sum to the highest value, or nil if nothing were found.

## Sample input and output

- The array [0, 1, 1, -1, 2, 3, 1] should return 4...6 because the highest sum of contiguous positive numbers is  $2 + 3 + 1$ , which lie at positions 4, 5, and 6 in the array.
- The array [10, 20, 30, -10, -20, 10, 20] should return 0...2.
- The array [1, -1, 2, -1] should return 2...2.
- The array [2, 0, 2, 0, 2] should return 0...0.
- The array [**Int**]() should return nil.

## Hints

**Hint #1:** This challenge is best solved using a little trial and error – start by writing tests to ensure your solution is good as you work.

**Hint #2:** Your return type should be **CountableClosedRange<Int>?** because there might not be any ranges of positive numbers.

**Hint #3:** This would be a good time to use **enumerated()** to retrieve items and their index from a collection.

**Hint #4:** There are two very different cases: if a number is greater than 0, and “everything else”. The index you return needs to be different, because a positive integer you find might be

the last one in the array.

## Solution

There might be clever solutions to this problem, but I went straight for a brute force approach in my own solution.

I solved it like this:

1. Create variables for the best range and best sum, as well as the start of the current range and the current sum.
2. Loop over the input numbers using `enumerated()` to get the index and number in each loop.
3. If the number is greater than zero, and the current range start is not nil, set the current range start to be the array index and add the number to the current sum.
4. If this new sum is better than the existing sum, update the best range to include this new number.
5. If we encounter a negative number, clear the current sum and current range start variables.

To make the code a little neater, I used nil coalescing to set the current range start integer to itself or the array index depending on whether it has a value already.

Here's my code:

```
func challenge50(_ numbers: [Int]) ->
CountableClosedRange<Int>? {
    // this is the return value, nil by default
    var bestRange: CountableClosedRange<Int>? = nil
    var bestSum = 0

    // these track the current sequence of positive integers
    var currentStart: Int? = nil
    var currentSum = 0
```

## Collections

```
for (index, number) in numbers.enumerated() {
    if number > 0 {
        // if we don't have a start for the current range, set
        one now
        currentStart = currentStart ?? index
        currentSum += number

        if currentSum > bestSum {
            // update our best range
            bestRange = currentStart! ... index
            bestSum = currentSum
        }
    } else {
        // reset the current range
        currentSum = 0
        currentStart = nil
    }
}

return bestRange
}
```

# Challenge 51: Reversing linked lists

**Difficulty:** Tricky

Expand your code from challenge 43 so that it has a **reversed()** method that returns a copy of itself in reverse.

**Tip:** Don't cheat! It is *not* a solution to this problem just to reverse the alphabet letters before you create your linked list. Create the linked list alphabetically, then write code to reverse it.

## Sample input and output

- When you call **reversed()** on your alphabet list, running **printNodes()** on the return value should print the English alphabet printed to the screen in reverse, i.e. “z y x … d b c a”.

## Hints

**Hint #1:** Most of the work is just producing a copy of the linked list.

**Hint #2:** Having to work on a copy makes this a little more interesting.

**Hint #3:** You could create two methods: one for copying, and one for reversing a copy in place. If you do this, please think carefully about Swift's naming conventions!

**Hint #4:** You need to create a **newNext** variable that starts as nil. Then traverse the full list, pull out its **next** value, then change the current node's **next** property to be **newNext**. You can then continue on to whatever node was in **next**, and repeat until the end of the list is reached.

## Solution

## Collections

This is such a simple task, and yet it's massive pain point for developers. On paper it's trivial: you can imagine in your head a series of boxes with arrows moving left to right, and it's your job to make the arrows move right to left. But in practice this can be devilish to solve, because for some reason your brain decides to go on a vacation rather than stick around and help you out.

The task is made doubly difficult by the fact that you almost certainly used a class for your linked list data types, and copying classes isn't automatic like you get with structs. Splitting the work into two methods has the advantage of making your code easier to understand and also more flexible in the future, but does mean going through your list twice.

Let's take a look at the all-in-one solution first:

```
func reversed() -> LinkedList<T> {
    // create our copy for the return value
    let copy = LinkedList<T>()

    // if we have nodes to copy...
    if let startNode = start {
        // copy the original start node
        var previousCopyNode = LinkedListNode(value:
            startNode.value)

        // now start copying from the next node
        var currentNode = start?.next

        while let node = currentNode {
            // create a copy of this node
            let copyNode = LinkedListNode(value: node.value)

            // make it point to the node we created previously
            copyNode.next = previousCopyNode
            previousCopyNode = copyNode
            currentNode = currentNode?.next
        }
    }
}
```

## Challenge 51: Reversing linked lists

```
// then make it the previous node, so we can move
forward
    previousCopyNode = copyNode
    currentNode = currentNode?.next
}

// we're at the end of the list, which is our new start
copy.start = previousCopyNode
}

return copy
}
```

That's a perfectly valid solution to this challenge, so we could easily stop here and move on. But I'd also like to explore with a smarter alternative: making a separate **copy()** method, then creating a **reverse()** method that reverses a linked list in place.

The official Apple way to implement copying is with the **NSCopying** protocol, which forces you to return **Any** from your copy method rather than, you know, a *useful* type. So, we'll skip that and just create our own **copy()** method, which is a modified version of the **reversed()** method I already showed you – it does the same thing, but leaves the nodes in their existing order.

Here's the code:

```
func copy() -> LinkedList<T> {
    let copy = LinkedList<T>()

    if let startNode = start {
        copy.start = LinkedListNode(value: startNode.value)
        var previousCopyNode = copy.start

        var currentNode = start?.next
    }
}
```

## Collections

```
        while let node = currentNode {
            let copyNode = LinkedListNode(value: node.value)
            previousCopyNode?.next = copyNode
            previousCopyNode = copyNode
            currentNode = currentNode?.next
        }
    }

    return copy
}
```

With that in place we can write a **reverse()** method that reverses a linked list in place. This is *much* simpler than copying and reversing at the same time:

```
func reverse() {
    var currentNode = start
    var newNext: LinkedListNode<T>? = nil

    while let node = currentNode {
        let next = node.next
        node.next = newNext
        newNext = node;
        currentNode = next
    }

    start = newNext
}
```

At this point we have a **copy()** method and a **reverse()** method, which means we can finally implement **reversed()** to satisfy the challenge:

```
func reversed() -> LinkedList<T> {
```

```
let copy = self.copy()
copy.reverse()
return copy
}
```

This approach took a little more thinking, and isn't quite as fast as in-place reversing because of the need to traverse the list twice, but a) you get to follow Swift's naming conventions (**reverse()** and **reversed()** match **sort()** and **sorted()**), and b) you have more flexibility in your linked list going forward, because you can write other methods that require a copy to be made without having to duplicate the copying all the time.

# Challenge 52: Sum an array of numbers

**Difficulty:** Tricky

Write one function that sums an array of numbers. The array might contain all integers, all doubles, or all floats.

## Sample input and output

- The code `challenge52(numbers: [1, 2, 3])` should return 6.
- The code `challenge52(numbers: [1.0, 2.0, 3.0])` should return 6.0.
- The code `challenge52(numbers: Array<Float>([1.0, 2.0, 3.0]))` should return 6.0.

## Hints

**Hint #1:** If this were just about counting integers this would definitely be an easy grade.

**Hint #2:** This function needs to work with multiple data types, so you'll need to use generics with a constraint.

**Hint #3:** You can solve this challenge functionally if you feel like impressing.

## Solution

This is a great question to ask someone who is interviewing for a mid-range to senior Swift position, because it requires them to know their protocols and generics fairly well.

Swift's **Numeric** protocol covers **Int**, **Float**, and **Double**, although it doesn't get much use. So, to solve this challenge you need to write a function that accepts an array of numbers that conform to **Numeric** and returns a single value:

## Challenge 52: Sum an array of numbers

```
func challenge52<T: Numeric>(numbers: [T]) -> T {  
}
```

Now it's just a matter of adding the array's numbers together. This can be done using a simple loop, like this:

```
var total: T = 0  
  
for number in numbers {  
    total = total + number  
}  
  
return total
```

Alternatively you can make it a functional one-liner using `reduce()`:

```
return numbers.reduce(0, +)
```

Done!

There is one partial alternative that's worth tossing into the mix, which is Apple's Accelerate framework. This is a massive library of optimized mathematics routines that is incredibly fast at doing selected tasks. In the context of this challenge, Accelerate is able to add doubles and floats extremely quickly – easily twice as fast as the fastest solution above, and perhaps four or five times as fast depending on what you feed it.

There are downsides to using Accelerate. First, it works on floats and doubles but not integers, so it's not suitable for the generic approach used here. Second, it works best when it has a lot of data: if you're summing only 10 numbers it will probably work out slower, but sum 100 or more and it will fly. Third, and most crucially, it is one of the least approachable APIs in all of Apple-land, so it takes a little getting used to.

Anyway, in case you were curious, here's the Accelerate way to sum an array of doubles:

## Collections

```
func challenge52c(numbers: [Double]) -> Double {
    var result: Double = 0.0
    vDSP_sveD(numbers, 1, &result, vDSP_Length(numbers.count))
    return result
}
```

# Challenge 53: Linked lists with a loop

**Difficulty:** Taxing

Someone used the linked list you made previously, but they accidentally made one of the items link back to an earlier part of the list. As a result, the list can't be traversed properly because it loops infinitely.

Your job is to write a method for your linked list that returns the node at the start of its loop, i.e. the one that is linked back to.

## Sample input and output

You can simulate a looped link list with this code:

```
var list = LinkedList<UInt32>()
var previousNode: LinkedListNode<UInt32>? = nil
var linkBackNode: LinkedListNode<UInt32>? = nil
var linkBackPoint = Int(arc4random_uniform(1000))

for i in 1...1000 {
    let node = LinkedListNode(value: arc4random())
    if i == linkBackPoint { linkBackNode = node }

    if let predecessor = previousNode {
        predecessor.next = node
    } else {
        list.start = node
    }
    previousNode = node
}
```

## Collections

```
}
```

```
previousNode?.next = linkBackNode
```

You will need to use whatever **LinkedList** and **LinkedListNode** structures you created in the previous challenge.

When your code has finished, your **findLoopStart()** should return the same node contained in the **linkBackNode** variable.

## Hints

**Hint #1:** There are two ways to solve this: using a set or using mathematics. You could also use an array, but only if you had no regard at all for performance.

**Hint #2:** If you take the set approach you will need to conform to **Hashable**, which in turn implies **Equatable**.

**Hint #3:** To conform to **Hashable** you must be able to give each of your linked list nodes a unique **hashValue** integer. You could store this as a property in your linked list, and increment it by 1 every time a `getUniqueHashValue()` method is called.

**Hint #4:** To conform to **Equatable** you will need to implement **static func ==** on your linked list node. This could be as simple as returning true if the two hash values are the same.

**Hint #5:** You can then loop over the nodes in your list, checking whether they are in a **seen** set. If a node isn't in the set, add it; if it *is* in the set, you have your loop point so return it. If you reach the end of your list it means you didn't have a loop, so return nil.

**Hint #6:** You can also solve this problem with pure mathematics, which is both significantly faster and more memory efficient. If you ever learned to do tortoise and hare loop detection, now is your chance to feel smug!

**Hint #7:** Your solution to challenge 44 provides the starting point for the mathematical solution here.

## Solution

This is a challenge that is easy if you took the right courses at university, but an exercise in yak shaving if you didn't. It's also an *extremely* common interview question, so it's one you should definitely prepare if you're serious.

Let's take a look at the naïve solution first, because it's possibly the one you reached for instinctively: create a **seen** set, then loop through all the nodes in your list and add them to that set until you find a duplicate. You probably started with a method similar to this:

```
func challenge53() -> LinkedListNode<T>? {
    // use our linked list's start node as the initial node to
    // read
    var currentNode = start
    var seen = Set<LinkedListNode<T>>()

    while let node = currentNode {
        if seen.contains(node) {
            return node
        } else {
            seen.insert(node)
            currentNode = node.next
        }
    }

    return nil
}
```

That starts at the beginning of the list, then goes through every node. If the **seen** set contains

## Collections

the node we can return it because we've found our match; if not we add it then move on. If we get to the end of the loop without returning it means no loop was detected, so we can return nil.

Yak shaving is a curious thing. The term comes from the idea that you find a task that needs to be done, such as changing a lightbulb, but while attempting that task you hit a problem, for example you realize you can't reach the light because your ladder is broken. So you decide to borrow your friend's ladder, but you realize you didn't finish reading that book she lent you three months ago. So you decide to finish reading the book so you can return it and get the ladder, but as you sit down on a chair to read the book you hear the chair squeak loudly, so you decide to get up to find some oil... and after an hour you find yourself at the local zoo shaving a yak, all so that you can change the lightbulb.

The code we just wrote is great, but only for definitions of "great" that don't include "working". You see, you can't add linked list nodes to a set, because they don't conform to **Hashable**. So you modify your class definition to this:

```
class LinkedListNode<T>: Hashable {
```

Done, right? Wrong. To conform to comparable you need to have a **hashValue** property that stores a unique integer, so you add it:

```
var hashValue: Int
```

And of course that means modifying the initializer too:

```
init(value: T, hashValue: Int) {
    self.value = value
    self.hashValue = hashValue
}
```

And *that* means finding a unique hash value in the first place so you can pass it to the initializer, so you'll need to add a property to the **LinkedList** class to store the highest hash value and a method to retrieve new ones:

## Challenge 53: Linked lists with a loop

```
private var uniqueHashValue = 0

func getUniqueHashValue() -> Int {
    uniqueHashValue += 1
    return uniqueHashValue
}
```

Having that in place means you can now call the new **LinkedListNode** initializer with a unique hash value, like this:

```
let node = LinkedListNode(value: arc4random(), hashValue:
list.getUniqueHashValue())
```

And *now* you're done. *Not.*

You see, you can't add things to a set without conforming to **Hashable**, but the **Hashable** protocol builds on the **Equatable** protocol, which means you need to conform to *that* too. To conform to **Equatable** you must implement **func ==** so Swift knows how to compare two instances of your type, so you need to add something like this to your **LinkedListNode** type:

```
static func ==(lhs: LinkedListNode, rhs: LinkedListNode) ->
Bool {
    return lhs.hashValue == rhs.hashValue
}
```

So, you added the **==** function to satisfy **Equatable**, added the **hashValue** to satisfy **Hashable**, added a new property and method to the **LinkedList** class, changed the **LinkedListNode** initializer, then changed the code to create your nodes... all so you can add nodes to a set and use its **contains()** method. *That's* yak shaving.

On the flip side, the yak is now shaved: the set solution I wrote way earlier now works as it ought to, so you could consider this challenge solved.

## Collections

However, there is a better solution – and by “better” I mean simpler, faster, and lighter on resources. This solution is called Floyd’s cycle-finding algorithm, sometimes called tortoise and hare, or fast runner / slow runner, and it’s actually a simple piece of mathematics genius. This isn’t CS101, but I love explaining things so here goes...

- We have a tortoise, who moves through the linked list one node at a time.
- We have a hare, who is faster than the tortoise, and moves through the linked list two nodes at a time.
- If there is a loop in the list, then sooner or later tortoise and hare must meet.
- They won’t necessarily meet at the start of the loop; they might meet in the middle of the loop. But soon or later, they *must* meet. If they don’t, there is no loop.
- Because the hare moves faster than the tortoise, it’s impossible for them to meet before the loop.
- When they eventually meet somewhere in the loop, you send the tortoise back to the start node again, and then start it moving forward one space again.
- This time, though, every time the tortoise takes one step forward, the hare also takes one step forward – rather than the previous two.
- The point at which they meet will be the start of your loop.

That’s the rough overview, but to explain how it works we’re going to need to use a bit of (hopefully non-threatening) algebra. Relax: I’ve tried to make it as clear as possible, so hopefully this should all make sense. This is all assuming there *is* a loop, mind you – if there isn’t, then we already returned nil and this is all very easy!

So, there’s a loop somewhere in our list, but we don’t know where. Before the loop starts, there’s a certain number of nodes that aren’t part of the loop. We don’t know how the number of nodes there are before the loop, so we’ll call that number  $B$ . At the end of  $B$  is where the loop starts, and it’s our job to find it.

By the time the tortoise arrives at the start of the loop, the hare will have already arrived and moved on. The hare might have gone around the loop completely and now be back at the start of the loop, or it might be anywhere between the start and the end of the loop, but it will be in

## Challenge 53: Linked lists with a loop

there somewhere, and it will eventually always meet the tortoise as the two of them go around.

When the tortoise and hare finally do meet, it will be a certain number of nodes into the loop. It might be zero (the start of the loop), 10, 1000, or whatever – we don't know. But they will eventually meet a certain number of steps into the loop, and we'll call that meeting number  $M$ .

So, when they meet, the tortoise will have moved  $B$  steps before the loop, plus  $M$  steps inside the loop, making  $B + M$  steps. Because the hare is moving two steps for every one step taken by the tortoise, it must have moved  $2 \times (B + M)$  steps, i.e. precisely twice as many steps as the tortoise.

Still with me? Chin up – we're half way there!

At this point, the tortoise has moved  $B + M$  steps, and the hare has moved  $2 \times (B + M)$  steps, and they met – they are at the same position in the linked list. This means the extra  $B + M$  steps moved by the hare was enough to bring it back to  $M$  in the loop.

If moving  $B + M$  steps always returns you back to the same position, then moving  $M$  steps fewer will leave you at  $B$ . So, all we need to do is find a way to make the hare move precisely  $B$  steps from the position where it met the tortoise and we'll have the start of the loop.

Now that the tortoise and hare have met, the algorithm changes. The hare stays in its position at  $M$  steps into the loop, but the tortoise goes back to the start of the linked list. They then resume moving, but this time they both move at the same speed: one square at a time.

The tortoise is repeating its steps, doing exactly what it did before, but now the hare is moving at the same speed. However, the hare started at  $M$ , which is the number of steps into the loop where it met the tortoise. We already know that moving  $B + M$  steps inside the loop leaves you where you started, but because the tortoise started at  $M$  it just needs to move  $B$  more steps to be at the start of the loop.

By the time the tortoise has moved  $B$  steps, the hare has also moved  $B$  steps. As I said already, after  $B$  steps the hare will be at the start of the loop. But  $B$  is also the number of steps from the start of the linked list to the start of the loop, and we've been moving the tortoise one step for

## Collections

every step the hare took.

As a result, after  $B$  steps have been taken, the tortoise and the hare will occupy the same position again, which means – at last – we can firmly identify the start of the loop: it's the point at which the tortoise and the hare meet for the second time.

Now, it's very possible you read all that and it gave you misty-eyed happy memories of computer science classes, but it's also possible – nay, likely – that you thought that all sounded terribly complicated and really you'd just like to see some code. Fortunately, writing the code for all that is a great deal easier than writing an explanation!

Here's the code:

```
func findLoopStartB() -> ListNode<T>? {
    var slow = start
    var fast = start

    // go through the list until we find the end
    while fast != nil && fast?.next != nil {
        // slow moves one space, fast moves two
        slow = slow?.next
        fast = fast?.next?.next

        // if the two met it means we found a loop, so exit the
        // loop
        if slow === fast {
            break
        }
    }

    // if fast or its successor is nil it means we made it to
    // the end of the list, so there's no loop
    guard fast != nil || fast?.next != nil else {
```

## Challenge 53: Linked lists with a loop

```
        return nil
    }

    // if we're still here, we know for sure there's a loop
    slow = start

    // loop through until we find another match
    while slow! != fast! {
        // move slow and fast the same speed now
        slow = slow?.next
        fast = fast?.next
    }

    // slow and fast now point to the same now, so return either
    one of them
    return slow
}
```

That code doesn't need any of the **Hashable** and **Equatable** complexity of the previous solution, plus it uses far less system resources and even runs about 6x faster.

I would even say that even if my explanation didn't quite make sense to you, the mathematics still hold true: start both at the beginning, move A forward one space and B forward two spaces until they meet, move A back to the beginning again, then move them both forward until they meet a second time – and you have the start of your loop.

Afterword: it's now 2:30am, and it took me four hours to write the tortoise and hare explanation thanks to multiple rewrites. If it comes across as incoherent, please believe me: I did try!

# Challenge 54: Binary search trees

**Difficulty:** Taxing

Create a binary search tree data structure that can be initialized from an unordered array of comparable values, then write a method that returns whether the tree is balanced.

**Tip #1:** There is more than one description of a balanced binary tree. For the purpose of this challenge, a binary tree is considered balanced when the height of both subtrees for every node differs by no more than 1.

**Tip #2:** Once you complete this challenge, keep your code around because you'll need it in the next one.

## Sample input and output

The following values should create balanced trees:

- [2, 1, 3]
- [5, 1, 7, 6, 2, 1, 9]
- [5, 1, 7, 6, 2, 1, 9, 1]
- [5, 1, 7, 6, 2, 1, 9, 1, 3]
- [50, 25, 100, 26, 101, 24, 99]
- ["k", "t", "d", "a", "z", "m", "f"]
- [1]
- [Character]()

The following values should not create balanced trees:

- [1, 2, 3, 4, 5]
- [10, 5, 4, 3, 2, 1, 11, 12, 13, 14, 15]
- ["f", "d", "c", "e", "a", "b"]

## Hints

**Hint #1:** You need to create a binary *search* tree rather than a plain binary tree. This means inserting nodes into the tree based on whether they are less than or equal (left) or greater than (right) their parent.

**Hint #2:** You should make your data types use a generic value that conforms to **Comparable**.

**Hint #3:** Each nodes should have a value, plus left and right optional nodes.

**Hint #4:** To find the correct place for each array item, start at the top of your tree then keep moving left or right until you find nil – that's your place.

**Hint #5:** You might find it useful to make your binary tree type conform to **CustomStringConvertible** so you can add a custom **var description: String** that prints the contents of your tree.

**Hint #6:** Checking a binary tree is balanced can be done by recursively comparing the minimum depth of both sides of a node against the maximum depth of both sides of a node. The tree can be considered balanced if the two values differ by no more than 1.

## Solution

Carl Sagan once said, “if you wish to make apple pie from scratch, you must first invent the universe.” That is, a simple task becomes more complicated when you realize there are other steps leading up to it.

In this challenge, you need to write a method that returns true if a binary tree is balanced, but that first means creating data types to represent a binary tree and its nodes, and also creating an initializer that converts an unsorted array of comparable values into tree nodes. If you were smart, you probably created some sort of way to visualize your tree to help you debug the initializer and depth counters.

## Collections

Let's start with the easy stuff: we can define a binary tree node as a generic class that stores a key, as well as left and right values. In code, it looks like this:

```
class Node<T> {
    var key: T
    var left: Node<T>?
    var right: Node<T>?

    init(key: T) {
        self.key = key
    }
}
```

There's no need to constrain the generic type, because that's the complete definition for **Node**. That class will be used to hold one node somewhere in the tree, but we can create another class that wraps up the whole tree:

```
class BinarySearchTree<T: Comparable> {
    var root: Node<T>?
}
```

So, the **BinarySearchTree** has one node, which is the root of the tree, and nothing else. It is another generic type, but this time with a constraint: the nodes it contains must store data that conforms to **Comparable**. This is required, because we need to place an item to the left of a node if it is less than or equal to it, or to the right of a node otherwise – we need to compare values.

Creating a binary search tree from an array of values isn't too tricky, and probably qualifies as "CS101":

1. Loop over every item in the array.
2. If we have a root node already, set it to be our tracker node – that's the one we're currently comparing against.

3. If the item is less than or equal to our tracker, and the tracker's left value is nil, make a new node from our item, make it the tracker's left value, then mark this item as being placed.
4. If the item is less than or equal to our tracker but we already have a left value, make that left value our new tracker and repeat the loop.
5. If the item is greater than our tracker, and tracker's right value is nil, make a new node from our item, make it the tracker's right value, then mark this item as being placed.
6. If the item is greater than our tracker but we already have a right value, make that right value our new tracker and repeat the loop.
7. If we don't already have a root node, make one from the item and make that the tracker.

That's it!

```
init(array: [T]) {
    for item in array {
        // this will be set to true when we've created a node
        // from this item
        var placed = false

        if let rootNode = root {
            // we have a root node, so make it our tracker
            var tracker = rootNode

            while placed == false {
                // if we're placing an item that comes before our
                // tracker
                if item <= tracker.key {
                    // if we don't already have a left node
                    if tracker.left == nil {
                        // make this item our left node
                        tracker.left = Node(key: item)

                        // mark it as placed
                    }
                }
            }
        }
    }
}
```

## Collections

```
        placed = true
    }

        // we already have a left node; make that the
tracker so we can compare against it.
        tracker = tracker.left!
    } else {
        // this item is greater than our tracker, so it
needs to sit on its right

        // if we don't already have a right node
if tracker.right == nil {
        // make this item our right node
        tracker.right = Node(key: item)

        // mark it as placed
        placed = true
    }

        // we already have a right node; make that the
tracker so we can compare against it.
        tracker = tracker.right!
    }
}

} else {
    root = Node(key: item)
}
}
```

The line `tracker = tracker.right!` contains a force unwrap, but it's OK because either `tracker.right` was nil and we set its value, or it was not nil and therefore has a value already –

either way, it will work.

At this point, we have a data structure that works and an initializer to create a tree from an array. You can check that it works by making the tree class conform to **CustomStringConvertible** like this:

```
class BinarySearchTree<T: Comparable>: CustomStringConvertible {
```

You can then create your own **description** property that prints the tree in whatever way you find useful. It's not required to fulfill this challenge, but it does make debugging a lot easier! Something like this would be a good place to start:

```
var description: String {
    guard let first = root else { return "(Empty)" }
    var queue = [Node<T>]()
    queue.append(first)

    var output = ""

    while queue.count > 0 {
        var nodesAtCurrentLevel = queue.count

        while nodesAtCurrentLevel > 0 {
            let node = queue.removeFirst()
            output += "\u{00a0}(node.key) "

            if node.left != nil { queue.append(node.left!) }
            if node.right != nil { queue.append(node.right!) }

            nodesAtCurrentLevel -= 1
        }
    }
}
```

## Collections

```
    output += "\n"
}

return output
}
```

That's all the hard stuff out of the way, but there's still one more part of the challenge: creating a method that returns true if the tree is balanced.

The simplest way to solve this problem is to start by finding the node with the minimum depth on either side of the tree. This is recursive: it starts at the top and works its way down. At the end of the tree we have a depth of one, and we can sum upwards from there.

For example, consider the input array [10, 5, 4, 3, 2, 1, 11, 12, 13, 14, 15], which is one of our test cases. To check whether the tree from that array is balanced, we'll start with 10, then go to the left: 5, 4, 3, 2, and finally 1. This part of the tree will only have left-hand nodes, because all the numbers are lower than their parent. So, 1 has a depth of 1 (it's the end of the tree), and 2 has a depth of  $1 + \max(\text{depths of children})$ . It only has one child, so it's choosing between children with a depth of 1 (its left-hand side), or 0 on its right-hand side, therefore the maximum is 1.

The process continues: 3 has a depth of  $1 + \max(\text{depths of children})$  (so 3), 4 has a depth of 4, and 5 has a depth of 5. We then move to the right and repeat, but this time it's all right-hand nodes: 15 has a depth of 1, 14 has a depth of 2, 13 has a depth of 3, 12 a depth of 4, and 11 a depth of 5. So, the left- and right-hand sides of our tree both have a maximum depth of 5, and we pass that back to the root element, which has a depth of  $1 + \max(\text{depths of children})$ , making 6.

The process is then repeated, but looking at the *minimum* depths. Again, it starts at 1, which is the end of the tree so it has a depth of 1. It then moves to 2, which has a depth of 1 on its left-hand side and a depth of zero (nothing) on its right-hand side. As we're now choosing the minimum depths, we'll use zero instead of 1, so we return  $1 + 0$  for this depth. This goes up to 3, which again does  $1 + 0$ , as do 4 and 5, so the total minimum depth for the left side of the tree

is 1. The situation on the right side of the tree is the same, so it also adds up to a minimum depth of 1. Finally, we get back to the root element, 10, which has a depth of  $1 + \text{the minimum depth of its children}$ , making 2.

Putting those two together, we find a maximum depth of 6 and a minimum depth of 2, and these numbers are allowed to differ by no more than 1 so we can safely say this tree is *not* balanced.

That might sound hard, but the code is remarkably trivial. Here it is, with a couple of `print()` statements inside so you can watch how it works on real data:

```
func isBalanced() -> Bool {
    func minDepth(from node: Node<T>?) -> Int {
        guard let node = node else { return 0 }
        let returnValue = 1 + min(minDepth(from: node.left),
minDepth(from: node.right))
        print("Got min depth \(returnValue) for \(node.key)")
        return returnValue
    }

    func maxDepth(from node: Node<T>?) -> Int {
        guard let node = node else { return 0 }
        let returnValue = 1 + max(maxDepth(from: node.left),
maxDepth(from: node.right))
        print("Got max depth \(returnValue) for \(node.key)")
        return returnValue
    }

    guard let root = root else { return true }

    let difference = maxDepth(from: root) - minDepth(from: root)
    return difference <= 1
}
```

## Collections

You could easily make the **minDepth()** and **maxDepth()** methods separate inside the class if you needed them for other purposes.

# **Chapter 5**

## Algorithms

# Challenge 55: Bubble sort

**Difficulty:** Easy

Create an extension for arrays that sorts them using the bubble sort algorithm.

**Tip:** A bubble sort repeatedly loops over the items in an array, comparing items that are next to each other and swapping them if they aren't sorted. This looping continues until all items are in their correct order.

## Sample input and output

- The array [12, 5, 4, 9, 3, 2, 1] should become [1, 2, 3, 4, 5, 9, 12].
- The array ["f", "a", "b"] should become ["a", "b", "f"].
- The array [String]() should become [].

## Hints

**Hint #1:** You'll need to extend the `Array` type, but only when its elements conform to `Comparable` so you can establish a sort order.

**Hint #2:** You want to repeat your loop while a condition is true, so `repeat` while makes sense.

**Hint #3:** Watch out for the case when the array is empty.

**Hint #4:** You can swap two values using the global `swap()` function like this:  
`array.swapAt(a, b)`.

**Hint #5:** If you try printing out the array after each sorting pass you might spot a pattern that you can use to optimize your code.

## Solution

Bubble sort is taught in many – if not most! – university computer science classes. Not because it's efficient (it really, really isn't) but because it's simple enough to explain and gives junior students a taste of a real algorithm.

If you've never tried to implement it before, this challenge might have felt more like a “taxing” grade, but hopefully you managed at least to implement a simple solution. Don't worry if you don't like your solution: there is no “good” bubble sort, only bad and worse.

Let's take a look at a simple solution first:

```
extension Array where Element: Comparable {
    func challenge55a() -> [Element] {
        // refuse to sort invalid arrays
        guard count > 1 else { return self }

        var returnValue = self
        var swapsMade: Bool

        repeat {
            // we're looping from scratch, so reset swapsMade to
false
            swapsMade = false

            // loop over all items in the array, excluding the
last one
            for index in 0 ..< returnValue.count - 1 {
                // pull out the current element and its successor
                let element = returnValue[index]
                let next = returnValue[index + 1]

                // if the current one is bigger than its successor...
                if (element > next) {

```

## Algorithms

```
// swap them
returnValue.swapAt(index, index + 1)

// and mark that we made a swap so the loop will
repeat
    swapsMade = true
}
}
}

} while swapsMade == true

// send back the sorted array
return returnValue
}

}
```

That's the simplest way I can think of implementing bubble sort, and hopefully you can see how it works.

With only a small change we can make this algorithm a little faster, and if you're not sure how try modifying the swapping check to this:

```
if (element > next) {
    print("Before \(returnValue)")
    returnValue.swapAt(index, index + 1)
    swapsMade = true
    print("After \(returnValue)")
}
```

That will show how the sort is operating on a given array. Here's part of the output you'll see:

```
Before [12, 5, 4, 9, 3, 2, 1]
After [5, 12, 4, 9, 3, 2, 1]
Before [5, 12, 4, 9, 3, 2, 1]
```

```

After [5, 4, 12, 9, 3, 2, 1]
Before [5, 4, 12, 9, 3, 2, 1]
After [5, 4, 9, 12, 3, 2, 1]
Before [5, 4, 9, 12, 3, 2, 1]
After [5, 4, 9, 3, 12, 2, 1]
Before [5, 4, 9, 3, 12, 2, 1]
After [5, 4, 9, 3, 2, 12, 1]
Before [5, 4, 9, 3, 2, 12, 1]
After [5, 4, 9, 3, 2, 1, 12]

```

Notice how the 12 rises immediately to the top? Keep in mind that the bubble sort algorithm moves linearly through the array, swapping as needed. When the highest item in the array is reached, in any position in the array, it will rise to the top quickly. First it will get swapped with the item on its right, then the loop will move forward one place to where the item is now, and it will be swapped again. The loop will move forward again, and it will be swapped *again*, and so on.

What this means is that after the array has been looped over once, you can be sure the first largest item is in its final place. After the loop has completed a second time, the second largest item is in its final place, and so on. In fact, after the loop has completed **N** times, the **Nth** largest item is in its final place.

This simple observation allows us to tweak our sort algorithm so that we loop from 0 up to the index of the highest sorted item – everything after that is guaranteed to be sorted already, so there's no need to keep checking.

Here's the optimized version:

```

func challenge55b() -> [Element] {
    guard count > 0 else { return [Element]() }

    var returnValue = self
    var highestSortedIndex = count

```

## Algorithms

```
repeat {
    var lastSwapIndex = 0

    for index in 0 ..< highestSortedIndex - 1 {
        let element = returnValue[index]
        let next = returnValue[index + 1]

        if (element > next) {
            returnValue.swapAt(index, index + 1)
            lastSwapIndex = index + 1
        }
    }

    highestSortedIndex = lastSwapIndex
} while highestSortedIndex != 0

return returnValue
}
```

That solution will still run slowly because ultimately it's limited by the general weakness of bubble sorting, but it's still an improvement of the previous version and hopefully easy enough for you to remember.

# Challenge 56: Insertion sort

**Difficulty:** Easy

Create an extension for arrays that sorts them using the insertion sort algorithm.

**Tip:** An insertion sort creates a new, sorted array by removing items individually from the input array and placing them into the correct position in the new array.

## Sample input and output

- The array [12, 5, 4, 9, 3, 2, 1] should become [1, 2, 3, 4, 5, 9, 12].
- The array ["f", "a", "b"] should become ["a", "b", "f"].
- The array [String]() should become [].

## Hints

**Hint #1:** You can perform insertion sort in-place, but that takes a little more thinking. Aim for correctness first, and efficiency later.

**Hint #2:** You will need to extend **Array** with a constraint on their elements so that they must be **Comparable** – that's what lets us sort items.

**Hint #3:** In the most simple solution, you should be able to pick out an item from your source array, then search through your sorted destination array to find where it should go.

**Hint #4:** If you want to try the in-place solution, pull out the current item you want to sort, then keep moving other elements to the right until you find the correct spot for your item.

## Solution

Insertion sort is much like bubble sort: it's generally considered a bad choice for a real-world

## Algorithms

search algorithm, but it's easy enough to teach. However, insertion sort is always faster than bubble sort, and in fact it's the preferred choice of algorithm if you're sorting small data sets – Swift's own `sort()` call uses it if your collection contains fewer than 20 items.

I'm going to walk you through two solutions: one that is simple enough that you ought to be able to memorize it if you want, and a second, more efficient solution that performs the operation in place.

First the easier solution. In this approach, we create a new, empty array that will be filled with sorted elements. Then we loop over every element in the unsorted array, and place each item individually into the sorted array. In the case where the sorted array is empty, we'll just use `append()`.

Here's the code:

```
extension Array where Element: Comparable {
    func challenge56a() -> [Element] {
        guard count > 1 else { return self }

        var returnValue = [Element]()

        for unsorted in self {
            if returnValue.count == 0 {
                returnValue.append(unsorted)
            } else {
                var placed = false

                for (index, sorted) in returnValue.enumerated() {
                    if unsorted < sorted {
                        returnValue.insert(unsorted, at: index)
                        placed = true
                        break
                    }
                }
            }
        }
    }
}
```

```
        }

        if !placed {
            returnValue.append(unsorted)
        }
    }

    return returnValue
}

}
```

I've tried to make that as simple as possible, so hopefully you can read it straight through. The key is in the inner loop: when we have a new item from the unsorted array, that loops through the sorted array and uses `enumerated()` and `<` to find the correct position.

Sorting the array in place is a little trickier, because you don't want to expand the array – you just want to move the items around to make space as you sort. Stick with me, though: this improved solution is about 2x faster than the previous one.

Here's how the improved algorithm works:

1. Take a copy of the original array. We're sorting in place now, so we don't want to modify the original.
  2. Loop through the array, starting position 1 (the second item) and continuing to the end.
  3. Take a copy of the item at the current index.
  4. Count backwards from the current position until either we've hit the start of the array.
  5. For each item that is greater than our current item, move it one place to the right in the array, and keep counting backwards.
  6. Finally, when we either reach the start of the array or an item that's greater than or equal to us, we place our item back into the array.

To explain this, imagine the following array: C, B, A. We'll count from position 1, which is B,

## Algorithms

so we'll put that into a temporary variable, and begin counting backwards. Going back one place we find that B is less than C (lexicographically), so we move C along one space to make the array C, C, A. Yes, C is in there twice.

At this point we hit the start of the array, so we can place our item – B – back in there, so the array is now B, C, A. The loop goes around again, and now A gets placed into a temporary variable. A is less than C, so C gets copied one place to the right to make B, C, C. A is also less than B, so that gets moved too, to make B, B, C. Finally, we hit the start of the array again, so A gets put there, making A, B, C – the array is sorted!

Here's all that in code:

```
func challenge56b() -> [Element] {
    guard count > 1 else { return self }

    var returnValue = self

    for i in 1 ..< count {
        var currentItemIndex = i

        // take a copy of the current item
        let itemToPlace = returnValue[currentItemIndex]

        // keep going around until we're at the start of the
        // array or find an item that's greater or equal to us
        while currentItemIndex > 0 && itemToPlace <
            returnValue[currentItemIndex - 1] {
            // move this item to the right
            returnValue[currentItemIndex] =
            returnValue[currentItemIndex - 1]
            currentItemIndex -= 1
        }
    }
}
```

## Challenge 56: Insertion sort

```
// place our item into its newly sorted place
returnValue[currentItemIndex] = itemToPlace
}

return returnValue
}
```

It takes a little more thinking, but you get such a performance boost it's worth doing. Remember, insertion sort is the preferred sorting algorithm for arrays with fewer than 20 items and is used by Swift itself – this is one that's worth getting right!

# Challenge 57: Isomorphic values

**Difficulty:** Easy

Write a function that accepts two values and returns true if they are isomorphic. That is, each part of the value must map to precisely one other, but that might be itself.

**Tip:** Strings **A** and **B** are considered isomorphic if you can replace all instances of each letter with another. For example, “tort” and “pump” are isomorphic, because you can replace both Ts with a P, the O with a U, and the R with an M. For integers you compare individual digits, so 1231 and 4564 are isomorphic numbers. For arrays you compare elements, so [1, 2, 1] and [4, 8, 4] are isomorphic.

## Sample input and output

These are all isomorphic values:

- “clap” and “slap”
- “rum” and “mud”
- “pip” and “did”
- “carry” and “baddy”
- “cream” and “lapse”
- 123123 and 456456
- 3.14159 and 2.03048
- [1, 2, 1, 2, 3] and [4, 5, 4, 5, 6]

These are not isomorphic values:

- “carry” and “daddy” – the Rs have become D, but C has also become D.
- “did” and “cad” – the first D has become C, but the second has remained D.
- “maim” and “same” – the first M has become S, but the second has become E.
- “curry” and “flurry” – the strings have different lengths.
- 112233 and 112211 – the number 1 is being mapped to 1, and the number 3 is also being

mapped to 1.

## Hints

**Hint #1:** Stringification holds the key to solve this problem simply. Your parameters should both be **Any**, and you can use the **String(describing:)** initializer to stringify them.

**Hint #2:** You need to loop over all the characters in both stringified character arrays. To avoid out of bounds problems, make sure you start by checking both strings are the same length.

**Hint #3:** You should store your character mappings using a dictionary of type **[Character: Character]**.

**Hint #4:** If you convert the characters of each string into an array you'll find subscripting significantly easier.

**Hint #5:** When you loop over each letter in the current string, you can check if it exists as a key in your character map. For characters that exist, check that its value matches the letter in the second string – if it doesn't, it's not an isomorphic string.

**Hint #6:** If your letter doesn't exist as a key in the character map, it's possible the second string's letter does exist as a value attached to a different key. If so, it's not an isomorphic string.

**Hint #7:** If the character isn't already a key, and the second string's letter isn't already a value, then add the character and matching second string letter to your character mapping dictionary.

**Hint #8:** If you've made it through all the characters in the first string and not encountered any problems, you have an isomorphic string.

## Solution

## Algorithms

If you were faced with this challenge in other languages, chances are it would target strings specifically. However, Swift has two advantages that many other languages do not: its **Any** type lets us represent any kind of data without needing generics, and its **String(describing:)** initializer lets us describe any kind of data using strings and characters.

The combination of these two features makes this challenge surprisingly straightforward to solve:

1. Convert both parameters to strings.
2. Ensure they are both the same length.
3. Create a character map dictionary that stores which input character maps to which output character.
4. Convert both strings to arrays of characters so that we can subscript them more easily.
5. Loop over every character in the first string and check whether it exists in our character map.
6. If it does exist and the existing mapping doesn't match the corresponding letter in the other string, return false.
7. If it doesn't exist in the character map, but the corresponding letter already has a mapping, return false because each letter should be mapped only once.
8. If we're still in the loop, it means both characters are new, so we set them in the dictionary.
9. Return true if we get to the end of the loop with all characters OK.

That's the answer. It's easy enough, I think, but certainly not simple – Swift does most of the heavy lifting for us with its **Any** to **String** conversion.

Here's my code:

```
func challenge57(first firstValue: Any, second secondValue: Any) -> Bool {  
    let first = String(describing: firstValue)  
    let second = String(describing: secondValue)  
  
    guard first.count == second.count else { return false }  
    var characterMap: [Character: Character] = [:]  
  
    for index in 0..        let firstChar = first[index]  
        let secondChar = second[index]  
  
        if characterMap[firstChar] != secondChar {  
            return false  
        }  
        if characterMap[firstChar] == nil {  
            characterMap[firstChar] = secondChar  
        }  
    }  
    return true  
}
```

```
var characterMap = [Character: Character]()
let firstArray = Array(first)
let secondArray = Array(second)

for (index, character) in firstArray.enumerated() {
    let otherCharacter = secondArray[index]

    if let currentMapping = characterMap[character] {
        if currentMapping != otherCharacter {
            return false
        }
    } else {
        if characterMap.values.contains(otherCharacter) {
            return false
        }
    }

    characterMap[character] = otherCharacter
}

return true
}
```

# Challenge 58: Balanced brackets

**Difficulty:** Easy

Write a function that accepts a string containing the characters (, [ , { , <, >, }, ], and ) in any arrangement and frequency. It should return true if the brackets are opened and closed in the correct order, and if all brackets are closed. Any other input should false.

## Sample input and output

- The string “()” should return true.
- The string “[[]]” should return true.
- The string “[[]](<{}>)” should return true.
- The string “[{}]{<[{}]>}” should return true.
- The string “” should return true.
- The string “} {” should return false.
- The string “[)]” should return false.
- The string “[)” should return false.
- The string “[<<<{}>>]” should return false.
- The string “hello” should return false.

## Hints

**Hint #1:** You should start by making the most simple check: does the string have only the eight different characters that are allowed?

**Hint #2:** Each type of opening bracket has only one matching opening bracket, so you should store that data somehow – a dictionary would seem sensible.

**Hint #3:** Every bracket need to be closed at some point, but not necessarily immediately – it might be closed many characters later, for example. So, you need to push it onto a stack, then

wait.

**Hint #4:** As you loop over each character in the string, it's either an opening bracket or a closing bracket. If it's an opening one it can go on your stack; if it's a closing one, then it should be the matching pair of whatever is on the end of your bracket stack.

**Hint #5:** If the function ends with anything left in the bracket stack it means there was one bracket that was not closed – a failure.

## Solution

This challenge requires a basic understanding of stacks: last in, first out (LIFO) collections that push items on to a data store then pop them off as needed. As each opening bracket comes in, we can push it onto the stack so that when it comes to reading back items the most recently opened bracket is the one we read first.

Swift doesn't have a native stack data type, but we can get the same behavior just by using an array and using its **append()** and **popLast()** methods.

First things first, though: ensuring that only brackets appear in the string. This can be done by creating a **CharacterSet** from the list of good letters (([{}<>])), then inverting the set and ensuring the input string doesn't match:

```
let validBrackets = "[{}<>]"
let validCharacterSet = CharacterSet(charactersIn:
validBrackets).inverted
guard input.rangeOfCharacter(from: validCharacterSet) == nil
else { return false }
```

Once that's done, we need to declare the relationships between opening and closing brackets, so Swift knows which closing bracket goes with each opening bracket. The easiest way to do that is with a dictionary, where the opening bracket is the dictionary key and the closing bracket is the value:

## Algorithms

```
let matchingBrackets: [Character: Character] = ["( ":" ")", "[ ":" "]",
  "{ ":" }", "< ":" >" ]
```

Using this approach, when we meet a character like ")" we can read the last used bracket from our stack, and check that **matchingBrackets[lastUsedBracket]** matches ")". If not, the brackets were closed out of order and the string is invalid.

Speaking of the stack, that's as simple as creating an array to store **Character** instances, like this:

```
var usedBrackets = [Character]()
```

If we make it to the end of the function without returning false – i.e., if all closing brackets matched their correct opening brackets – then we can return true if the stack is empty, like this:

```
return usedBrackets.count == 0
```

Done! Here's my solution in full, with extra comments:

```
func challenge58(input: String) -> Bool {
    let validBrackets = "([{<>}])"
    let validCharacterSet = CharacterSet(charactersIn:
validBrackets).inverted
    guard input.rangeOfCharacter(from: validCharacterSet) == nil
else { return false }

    let matchingBrackets: [Character: Character] = ["( ":" ")",
"[ ":" "]", "{ ":" }", "< ":" >" ]
    var usedBrackets = [Character]()

    for bracket in input {
        if matchingBrackets.keys.contains(bracket) {
            // this is an opening bracket
            usedBrackets.append(bracket)
        } else {
            if let matchingBracket = matchingBrackets[bracket] {
                if usedBrackets.last != matchingBracket {
                    return false
                }
                usedBrackets.removeLast()
            } else {
                return false
            }
        }
    }
    return usedBrackets.isEmpty
}
```

## Challenge 58: Balanced brackets

```
        usedBrackets.append(bracket)
    } else {
        // this is a closing bracket – try to pull off our
previous open
        if let previousBracket = usedBrackets.popLast() {
            if matchingBrackets[previousBracket] != bracket {
                // if they don't match, this is a bad string
                return false
            }
        } else {
            // we don't have an opening bracket, this is a bad
string
            return false
        }
    }
}

return usedBrackets.count == 0
}
```

# Challenge 59: Quicksort

**Difficulty:** Tricky

Create an extension for arrays that sorts them using the quicksort algorithm.

**Tip #1:** The quicksort algorithm picks an item from its array to use as the pivot point, then splits itself into either two parts (less than or greater than) or three (less, greater, or equal). These parts then repeat the pivot and split until the entire array has been split, then it gets rejoined so that less, equal, and greater are in order.

**Tip #2:** I can write quicksort from memory, but I cannot write *fully optimized* quicksort from memory. It's a complex beast to wrangle, and it requires careful thinking – honestly, I have better things to keep stored in what little space I have up there! So, don't feel bad if your attempt is far from ideal: there's no point creating a perfect solution if you struggle to remember it during an interview.

**Tip #3:** Quicksort is an algorithm so well known and widely used that you don't even write a space in its name – it's “quicksort” rather than “quick sort”.

## Sample input and output

- The array [12, 5, 4, 9, 3, 2, 1] should become [1, 2, 3, 4, 5, 9, 12].
- The array ["f", "a", "b"] should become ["a", "b", "f"].
- The array [String]() should become [].

## Hints

**Hint #1:** You will need to extend **Array** with a constraint on their elements so that they must be **Comparable** – that's what lets us sort items.

**Hint #2:** There are lots of ways to pick a pivot point; choosing a random item is probably best.

**Hint #3:** You can do most of the work with `filter()` if you want.

**Hint #4:** For maximum performance, try to solve the challenge in-place using `inout`.

## Solution

Quicksort is an incredibly slow algorithm when implemented poorly, and an incredibly fast algorithm when implemented excellently. As you might imagine, the complexity of your code increases as the speed does, and in some respects making a flexible, re-usable form of quicksort feels like trying to run a marathon with your legs tied together.

The requirement for this challenge was to extend arrays, which means you need to have used a method inside a constrained extension. So, let's look at what I consider to be the ideal solution to this challenge:

```
extension Array where Element: Comparable {
    func challenge59a() -> [Element] {
        guard count > 1 else { return self }

        // pivot on the center of the array
        let pivot = self[count / 2]

        // create three new buckets that we'll sort ourselves
        into
        var before = [Element]()
        var after = [Element]()
        var equal = [Element]()

        // loop over all items, placing each one into a bucket by
        comparing against our pivot
        for item in self {
            if item < pivot {

```

## Algorithms

```
        before.append(item)
    } else if item > pivot {
        after.append(item)
    } else {
        equal.append(item)
    }
}

// call this function recursively then return the
combined input
return before.challenge59a() + equal +
after.challenge59a()
}
```

That's a pretty poor implementation of quicksort in terms of raw performance, but it certainly passes the challenge and ought to be simple enough to memorize.

A slightly slower implementation that's even easier to memorize looks like this:

```
func challenge59b() -> [Element] {
    guard count > 1 else { return self }

    let pivot = self[count / 2]
    let before = filter { $0 < pivot }
    let after = filter { $0 > pivot }
    let equal = filter { $0 == pivot }

    return before.challenge59b() + equal + after.challenge59b()
}
```

The nice thing about this solution – ignoring the repeated work being done by the calls to `filter()` – is that it's crystal clear how quicksort works. In fact, that code is only a little different

to some of the (many!) quicksort-in-Swift implementations that were being posted just days after Swift was originally announced – that's how popular quicksort is!

Now, I want to talk briefly about performance, which I realize is outside the remit of this book but it *is* important here because quicksort is sensitive to small changes.

First, choosing a pivot point. In the two solutions above I used **self[count / 2]**, however another popular choice is this:

```
let pivot = self[Int(arc4random_uniform(UInt32(count)))]
```

Yes, choosing a completely random pivot. And yes, that's a perfectly sensible thing to do.

The truth is that picking a pivot point is *complicated*, and in fact the way you choose a pivot really depends on your array. For example, if your array is often already sorted, using **count / 2** is a great choice. If your array is very large, then you'll want to take a more complex approach because the time spent picking a better pivot is going to pay off with reduced sort time. So, you might choose something like median of three, where you pick three random elements from the array and pivot on whichever one lies in the center.

The reason for choosing a random pivot point is that quicksort's performance varies depending on its data. If you had the array [1, 2, 500, 2, 1] and you pivoted on the center, 500, you'll do a lot of work for almost no gain. Choosing a random pivot point every time means that you minimize the chance of consistently choosing a bad pivot point, which helps quicksort perform well overall.

Once you've chosen a good pivot point, how can you make quicksort fast? Well, it's *massively* outside the remit of this book to discuss the extensive research that's been done into quicksort, but I can at least walk you through what is probably the most optimized form of quicksort I can write from memory. It's not the fastest out there, but it's still about 25x faster than the solutions above while still being more or less readable – and still passing this challenge.

Let me walk you through how it works:

## Algorithms

1. The function works entirely in place, so it's a **mutating** method.
2. It's called using left and right parameters, which marks the array start and end position that's being sorted. This will initially be 0 and the array length - 1.
3. It picks a pivot at the end of the array, and also creates a variable that marks the point in the array that will store where items are greater than the pivot.
4. It then loops through the array, counting from **left** to **right**. Again, initially that's 0 and the array length - 1.
5. If the current item being scanned is less than the pivot, it swaps them item with whatever is at the split point, then moves the split point up one place.
6. Move the **right** parameter (that's the one being used as our pivot) to the split point.
7. Finally, call itself twice more, passing in the left-hand side first, then the right-hand side second.

That's it. Like I said, you could make a faster one, but there's always a trade-off between "that's fast" and "I can remember that."

Here's the code, first without comments so you can see how short it is:

```
extension Array where Element: BinaryInteger {  
    mutating func challenge59c(left: Int, right: Int) {  
        guard left < right else { return }  
        let pivot = self[right]  
        var splitPoint = left  
  
        for i in left ..< right {  
            if self[i] < pivot {  
                (self[i], self[splitPoint]) = (self[splitPoint],  
                self[i])  
                splitPoint += 1  
            }  
        }  
    }  
}
```

## Challenge 59: Quicksort

```
        (self[right], self[splitPoint]) = (self[splitPoint],
self[right])
        challenge59c(left: left, right: splitPoint - 1)
        challenge59c(left: splitPoint + 1, right: right)
    }
}
```

And here it is with comments, so hopefully you can see exactly how it fits together:

```
extension Array where Element: BinaryInteger {
    mutating func challenge59c(left: Int, right: Int) {
        // make sure we have a sensible range to work with
        guard left < right else { return }

        // use the right-hand element, because that's moved last
        let pivot = self[right]

        // set our split point – the marker where elements start
        // being greater than the pivot – to be the left edge
        var splitPoint = left

        // count through all items in the array
        for i in left ..< right {
            // if this item is less than our pivot
            if self[i] < pivot {
                // move it so that it's at the split point
                (self[i], self[splitPoint]) = (self[splitPoint],
self[i])

                // then move the split point forward one place
                splitPoint += 1
            }
        }
    }
}
```

## Algorithms

```
// move our pivot item to the split point
(self[right], self[splitPoint]) = (self[splitPoint],
self[right])

        // recursively call this function on everything before
the split...
challenge59c(left: left, right: splitPoint - 1)

        // ...and everything after the split
challenge59c(left: splitPoint + 1, right: right)
}

}
```

There you go – a quicksort implementation that’s suitably fast, and hopefully also easy enough for you to remember.

# Challenge 60: Tic-Tac-Toe winner

**Difficulty:** Tricky

Create a function that detects whether either player has won in a game of Tic-Tac-Toe.

**Tip:** A tic-tac-toe board is 3x3, containing single letters that are either X, O, or empty. A win is three Xs or Os in a straight line.

## Sample input and output

- The array `[["X", "", "O"], ["", "X", "O"], ["", "", "X"]]` should return true.
- The array `[["X", "", "O"], ["X", "", "O"], ["X", "", ""]]` should return true.
- The array `[["", "X", ""], ["O", "X", ""], ["O", "X", ""]]` should return true.
- The array `[["", "X", ""], ["O", "X", ""], ["O", "", "X"]]` should return false.
- The array `[["", "", ""], ["", "", ""], ["", "", ""]]` should return false.

## Hints

**Hint #1:** Your board parameter should be `[[String]]` – an array of array of strings.

**Hint #2:** You can evaluate the rows and columns in a loop.

**Hint #3:** You can evaluate diagonals using two checks: one from top left to bottom right, and one from bottom left to top right.

**Hint #4:** You might want to use a nested function to make your code cleaner.

## Solution

I think the best way to solve this challenge is incrementally: start with a simple solution that works correctly but could be more elegant, then improve it to remove duplicated code.

## Algorithms

For this challenge, the basic solution looks like this:

```
func challenge60a(_ board: [[String]]) -> Bool {
    for i in 0 ..< 3 {
        // check each row
        if board[i][0] != "" && board[i][0] == board[i][1] &&
board[i][0] == board[i][2] {
            return true
        }

        // check each column
        if board[0][i] != "" && board[0][i] == board[1][i] &&
board[0][i] == board[2][i] {
            return true
        }
    }

    // now check diagonally top left to bottom right
    if board[0][0] != "" && board[0][0] == board[1][1] &&
board[0][0] == board[2][2] {
        return true
    }

    // and check diagonally bottom left to top right
    if board[0][2] != "" && board[0][2] == board[1][1] &&
board[0][2] == board[2][0] {
        return true
    }

    // if we're still here there's no winner
    return false
}
```

To make myself quite clear: that's a great solution to the challenge. Seriously, if you got asked this question at an interview and you wrote the above, you've done very well.

Could it be better? Sure – and if you have time left on the clock, you could have another pass to improve what you have. Specifically, each of the checks for a win involve repeating one square three times: “is square X empty?”, “is square X equal to square Y?” and “is square X equal to square Z?”

To clean this up, and in doing so make the code much more readable, we can create a nested `isWin()` function that returns true if all three squares contain a move from the same player.

So, this would be a sensible second pass for this challenge:

```
func challenge60b(_ board: [[String]]) -> Bool {
    func isWin(_ first: String, _ second: String, _ third: String) -> Bool {
        guard first != "" else { return false }
        return first == second && first == third
    }

    for i in 0 ..< 3 {
        if isWin(board[i][0], board[i][1], board[i][2]) {
            return true
        }

        if isWin(board[0][i], board[1][i], board[2][i]) {
            return true
        }
    }

    if isWin(board[0][0], board[1][1], board[2][2]) {
        return true
    }
}
```

## Algorithms

```
}

if isWin(board[0][2], board[1][1], board[2][0]) {
    return true
}

return false
}
```

With that change, it becomes clear that the final three **return** statements can be collapsed into one using `||`: if either the first one or the second check is a winning move then they will return true, otherwise false.

So, a third iteration of this solution would look like this:

```
func challenge60c(_ board: [[String]]) -> Bool {
    func isWin(_ first: String, _ second: String, _ third: String) -> Bool {
        guard first != "" else { return false }
        return first == second && first == third
    }

    for i in 0 ..< 3 {
        if isWin(board[i][0], board[i][1], board[i][2]) {
            return true
        }

        if isWin(board[0][i], board[1][i], board[2][i]) {
            return true
        }
    }

    return isWin(board[0][0], board[1][1], board[2][2]) ||
}
```

## Challenge 60: Tic-Tac-Toe winner

```
isWin(board[0][2], board[1][1], board[2][0])  
}
```

If you intend to iterate on your solutions like this, notice that I name my solutions “a”, “b”, and “c”. What you *don’t* want to do is modify your code in place then have nothing to show when your time runs out because you were part-way through something!

# Challenge 61: Find prime numbers

**Difficulty:** Tricky

Write a function that returns an array of prime numbers from 2 up to but excluding  $N$ , taking care to be as efficient as possible.

**Tip:** Calculating primes is easy. Calculating primes *efficiently* is not. Take care!

## Sample input and output

- The code `challenge61(upTo: 10)` should return 2, 3, 5, 7.
- The code `challenge61(upTo: 11)` should return 2, 3, 5, 7; remember to exclude the upper bound.
- The code `challenge61(upTo: 12)` should return 2, 3, 5, 7, 11.

## Hints

**Hint #1:** Writing code to find whether one number is prime or not is very different to writing code to find *all* prime numbers – there’s a reason this is in the algorithms chapter.

**Hint #2:** When given a number, you decide whether it’s prime by checking whether it has any factors. When given a range of numbers, you want to take the opposite approach: assume all numbers are prime, then remove numbers that are composites by multiplying primes.

**Hint #3:** This is known as the Sieve of Eratosthenes: take the number 2, then mark all multiples of 2 as being not prime. Then take the number 3 and repeat. Then 5 (no need to check 4; that’s a multiple of 2), then 7 (no need to check 6; that’s a multiple of 3), and so on. What remains must be prime.

**Hint #4:** Once you have an array containing which numbers are prime and which are not, you

just need to extract the numbers that are prime and return them.

## Solution

The Sieve of Eratosthenes is an algorithm I teach to children because it's so smart it feels like cheating. Not only does it run screamingly fast, but it's easy enough you can explain in under a minute and remember for the rest of your life. I also lead a secret life as a classical studies student, so its ancient origins are especially appealing to me!

Here it is: mark your entire range of numbers as being prime, so let's say that's 0 to 10. We know that 0 and 1 can't be prime by definition, so we mark those as not prime. Now we loop from 2 up to the maximum of our range: if that number is currently marked prime, then we can mark all its multiples as not prime. So, 2 is prime, which means 4, 6, and 8 are not, so we mark them as not prime. We then continue to the next number, which is 3, and mark its multiples as not prime: 6 and 9. We then continue to 4, but it's already been marked as not prime so we can continue to 5, and so on.

That's it. That's the entire algorithm, and even though it's over 2000 years old it still remains one of the most efficient ways to find ranges of prime numbers – so it's the perfect solution to this challenge.

As always I've provided my solution below, but it bears a little explanation. My sieve array is declared as this:

```
var sieve = [Bool](repeating: true, count: max)
```

If **max** were 10, that would create an array of 10 elements, all set to **true** – we mark all numbers as being prime to start with. I then mark positions 0 and 1 as being false, because 0 and 1 aren't prime by definition. So, my array doesn't contain the numbers itself. Instead, it contains whether the number at that index is prime, so **sieve[3]** should ultimately be true because 3 is prime.

Flipping booleans between true and false is faster than removing items from an array, but isn't

## Algorithms

suitable for the return value in this challenge. So, we need to convert the boolean array into an integer array, including only numbers that are prime.

Because my boolean array uses index to identify each number, what we want to do is return the index of any boolean that is true. Fortunately, that's trivial by combining `enumerated()` with `compactMap()`: the former gives us both the index and boolean value for each element, and the latter can be used to convert “index of item or nil” into an array of integers because it strips out nils.

With all that in mind, here's my solution:

```
func challenge61(upTo max: Int) -> [Int] {
    guard max > 1 else { return [] }

    var sieve = [Bool](repeating: true, count: max)
    sieve[0] = false
    sieve[1] = false

    for number in 2 ..< max {
        if sieve[number] == true {
            for multiple in stride(from: number * number, to:
sieve.count, by: number) {
                sieve[multiple] = false
            }
        }
    }

    return sieve.enumerated().compactMap { $1 == true ? $0 :
nil }
}
```

How fast is it? Well, if you compile it in fully optimized mode, it can calculate all the primes up to 10 million in about 0.05 seconds on a modern Mac – not bad!

## Challenge 61: Find prime numbers

**Note:** Remember that prime numbers can never be even, because even numbers are always divisible by two. Can you optimize this code further to skip even numbers?

# Challenge 62: Points to angles

**Difficulty:** Tricky

Write a function that accepts an array of **CGPoint** pairs, and returns an array of the angles between each point pair. Return the angles in degrees, where 0 or 360 is straight up.

**Tip:** If it helps your thought process, imagine each point pair as being two touches on the screen: you have the first touch and the second, what is the angle between them?

## Sample input and output

Here is some code you can test with:

```
var points = [(first: CGPoint, second: CGPoint)]()
points.append((first: CGPoint.zero, second: CGPoint(x: 0, y: -100)))
points.append((first: CGPoint.zero, second: CGPoint(x: 100, y: -100)))
points.append((first: CGPoint.zero, second: CGPoint(x: 100, y: 0)))
points.append((first: CGPoint.zero, second: CGPoint(x: 100, y: 100)))
points.append((first: CGPoint.zero, second: CGPoint(x: 0, y: 100)))
points.append((first: CGPoint.zero, second: CGPoint(x: -100, y: 100)))
points.append((first: CGPoint.zero, second: CGPoint(x: -100, y: 0)))
print(challenge62(points: points))
```

If your code has worked correctly, that should print [0.0, 45.0, 90.0, 135.0, 180.0, 225.0, 270.0, 315.0]. Returning 360.0 for the first number is also acceptable.

## Hints

**Hint #1:** You’re mapping an array of one type to an array of another type. Yes, that means `map()` is a good idea.

**Hint #2:** You’ll need to use `atan2()` to calculate the angle between any two points, providing the Y delta and the X delta as its two parameters.

**Hint #3:** `atan2()` works in radians, so you’ll need to convert the value to degrees using the formula `degrees = radians * 180 / Double.pi`.

**Hint #4:** Where “0” lies is of course arbitrary, but the convention is that 0 degrees runs along the positive X axis. You need to correct for this.

## Solution

This is the kind of work you might once have thought you left behind at university, but positions and angles form the bedrock of advanced touch handling on iOS. It’s common to see code like this when working with `UIPanGestureRecognizer`:

```
let velocity = recognizer.velocity(in: self.view)
var angle = atan2(velocity.y, velocity.x) * 180.0 / CGFloat.pi
if (angle < 0) { angle += 360.0 }
```

The adjustment at the end ensures the angles lies in the range 0 to 360. This challenge requires you to do that across an array of points, while also compensating for the 90-degree offset.

Here’s my solution:

## Algorithms

```
func challenge62(points: [(first: CGPoint, second: CGPoint)]) -> [CGFloat] {
    return points.map {
        let radians = atan2($0.first.y - $0.second.y, $0.first.x - $0.second.x)
        var degrees = (radians * 180 / CGFloat.pi) - 90
        if (degrees < 0) { degrees += 360.0 }
        if degrees == 360 { degrees = 0 }
        return degrees
    }
}
```

Note: I added an **if degrees == 360** check in there so that the function returns 0 rather than 360, but that's just personal preference – it's not required.

# Challenge 63: Flood fill

**Difficulty:** Taxing

Write a function that accepts a two-dimensional array of integers that are 0 or 1, a new number to place, and a position to start. You should read the existing number at the start position, change it to the new number, then change any surrounding numbers that matched the start number, then change any surrounding *those*, and so on - like a flood fill algorithm in Photoshop.

**Tip #1:** If you value your sanity, you will add **import GameplayKit** then generate your grid using this code:

```
let random = GKmersenneTwisterRandomSource(seed: 1)
var grid = (1...10).map { _ in (1...10).map { _ in
    Int(random.nextInt(upperBound: 2)) } }
```

That will allow you to produce the same grid every time, which ought to make debugging easier. (Note: I made **grid** a variable rather than a constant for a reason.)

**Tip #2:** A flood fill works by filling grid positions directly above, below, to the left, and to the right, stopping only when a different number is encountered.

**Tip #3:** If the arrays contained all zeros, filling 5 would cause the arrays to contain all 5s because all numbers would be filled.

## Sample input and output

Given the following set up:

```
let random = GKmersenneTwisterRandomSource(seed: 1)
let grid = (1...10).map { _ in (1...10).map { _ in
    Int(random.nextInt(upperBound: 2)) } }
```

## Algorithms

You will have the following grid:

```
[0, 0, 0, 0, 0, 1, 0, 0, 1, 1]
[0, 1, 1, 0, 0, 0, 0, 1, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 1, 1]
[1, 0, 1, 0, 0, 1, 1, 0, 0, 0]
[1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
[1, 0, 1, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 1, 1, 0, 1, 1, 1]
[1, 1, 1, 0, 0, 1, 1, 1, 1, 1]
[1, 1, 0, 1, 1, 1, 1, 0, 0, 0]
[0, 1, 1, 0, 0, 1, 0, 1, 1, 1]
```

After running this code:

```
challenge63(fill: 5, in: grid, at: (x: 2, y: 0))
```

You will have the following grid:

```
[5, 5, X, 5, 5, 1, 5, 5, 1, 1]
[5, 1, 1, 5, 5, 5, 5, 1, 0, 0]
[5, 1, 5, 5, 5, 5, 5, 5, 1, 1]
[1, 0, 1, 5, 5, 1, 1, 5, 5, 5]
[1, 0, 1, 5, 1, 1, 1, 1, 1, 5]
[1, 0, 1, 1, 5, 5, 5, 5, 5, 5]
[0, 0, 0, 0, 1, 1, 1, 5, 1, 1]
[1, 1, 1, 0, 0, 1, 1, 1, 1, 1]
[1, 1, 0, 1, 1, 1, 0, 0, 0, 0]
[0, 1, 1, 0, 0, 1, 0, 1, 1, 1]
```

Note, in the above grid I marked with X where the fill started, but that will be a five too.

## Hints

**Hint #1:** You can solve this using a recursive function, or using a loop.

**Hint #2:** You will almost certainly find it useful to add `print()` statements inside your loop or function that displays what square is being manipulated, and perhaps also the current grid.

**Hint #3:** Keep a stack of squares that need to be filled. Start by adding your initial square, then loop over that and remove one square to fill / spread from each time.

## Solution

This is not a difficult problem to solve, but *is* difficult to solve cleanly and efficiently, and, most important of all, in a way that can be explained clearly to someone else.

Like I said in the hints, you could solve this using a loop or using a recursive function. I'll show you both, but for pedagogical reasons I think the recursive function is significantly better. That's not to say the looping option is *wrong*, just that my job is to build a solution that's useful for teaching purposes, and by that measure I think the recursive option is significantly easier to understand.

Let's start with the looping option. I've solved it this way using a nested function, because it makes the code fractionally nice. Here's how it works:

1. Take a copy of the original grid.
2. Create an array of X/Y co-ordinates that need to be filled, giving it the initial start point to begin with.
3. Pull out the current number at the start position, so we know what numbers we're filling.
4. While there are still items in the squares to fill array, pull out one and fill it.
5. Now add any adjacent squares to the array, but only if they are within the grid and have the same starting number we used for the fill.

That's it. I made step 5 into a nested function that uses closure capturing for the grid array and

## Algorithms

start number because it made the code both neater and easier to understand. Here's the code:

```
func challenge63a(fill: Int, in grid: [[Int]], at point: (x: Int, y: Int)) -> [[Int]] {
    var returnValue = grid
    var squaresToFill = [point]
    let startNumber = grid[point.y][point.x]

    func tryAddMove(_ move: (x: Int, y: Int)) {
        guard move.x >= 0 else { return }
        guard move.x < 10 else { return }
        guard move.y >= 0 else { return }
        guard move.y < 10 else { return }
        guard returnValue[move.y][move.x] == startNumber else
        { return }
        squaresToFill.append(move)
    }

    while let square = squaresToFill.popLast() {
        guard returnValue[square.y][square.x] != fill else
        { continue }
        returnValue[square.y][square.x] = fill

        tryAddMove((x: square.x, y: square.y - 1))
        tryAddMove((x: square.x, y: square.y + 1))
        tryAddMove((x: square.x - 1, y: square.y))
        tryAddMove((x: square.x + 1, y: square.y))
    }

    return returnValue
}
```

While that does solve the problem, it might be hard to follow. You can scatter calls to **print()**

throughout to watch the algorithm work its way across the board, like this:

```
func challenge63a(fill: Int, in grid: [[Int]], at point: (x: Int, y: Int)) -> [[Int]] {
    var returnValue = grid
    var squaresToFill = [point]
    let startNumber = grid[point.y][point.x]

    func tryAddMove(_ move: (x: Int, y: Int)) {
        guard move.x >= 0 else { return }
        guard move.x < 10 else { return }
        guard move.y >= 0 else { return }
        guard move.y < 10 else { return }
        guard returnValue[move.y][move.x] == startNumber else
        { return }

        print("\u{1f49}(move) is currently \u{1f49}(returnValue[move.y][move.x])")

        squaresToFill.append(move)
    }

    print("BEFORE")
    returnValue.forEach { print($0) }
    print("")

    while let square = squaresToFill.popLast() {
        guard returnValue[square.y][square.x] != fill else
        { continue }
        print("Filling \u{1f49}(square) with \u{1f49}(fill)")
        returnValue[square.y][square.x] = fill

        print("****")
    }
}
```

## Algorithms

```
returnValue.forEach { print($0) }

tryAddMove((x: square.x, y: square.y - 1))
tryAddMove((x: square.x, y: square.y + 1))
tryAddMove((x: square.x - 1, y: square.y))
tryAddMove((x: square.x + 1, y: square.y))

}

print("AFTER")
returnValue.forEach { print($0) }

return returnValue
}
```

That's the same thing, just with calls to `print()` before, during, and after the fill – you should be able to see squares being filled individually as the loop goes around.

Another way to solve the same problem is using a recursive function call: you fill a square, which fills the squares around it, which fill the squares around *them*, and so on. I made `grid` a variable rather than a constant for this very reason: we can make `grid` an `inout` parameter, thus enabling all the function calls to modify the same grid.

This code is a great deal simpler than the looping option, because there's no need for the nested function any more. Instead, I've made it accept an optional final parameter, `replacing`, which is the number to replace. The first time the function is run that won't be provided, so we'll read it straight from the grid. But on subsequent runs – when the function calls itself – we need to pass that in, because the `point` parameter will be replaced by the new position to fill.

Here's the recursive solution:

```
func challenge63b(fill newNumber: Int, in grid: inout [[Int]],  
at point: (x: Int, y: Int), replacing: Int? = nil) {
```

```
// bail out if this position is invalid
guard point.x >= 0 else { return }
guard point.x < 10 else { return }
guard point.y >= 0 else { return }
guard point.y < 10 else { return }

// `replacing` will be set on 2nd and subsequent runs
let startNumber = replacing ?? grid[point.y][point.x]

if grid[point.y][point.x] == startNumber {
    // adjust this point
    grid[point.y][point.x] = newNumber

    // and fill in surrounding squares
    challenge63b(fill: newNumber, in: &grid, at: (x: point.x,
y: point.y - 1), replacing: startNumber)
    challenge63b(fill: newNumber, in: &grid, at: (x: point.x,
y: point.y + 1), replacing: startNumber)
    challenge63b(fill: newNumber, in: &grid, at: (x: point.x
- 1, y: point.y), replacing: startNumber)
    challenge63b(fill: newNumber, in: &grid, at: (x: point.x
+ 1, y: point.y), replacing: startNumber)
}
```

Even with comments added that's still a shorter solution than using a loop, and I think more elegant too. You be the judge!

# Challenge 64: N Queens

**Difficulty:** Taxing

There are  $M$  different ways you can place  $N$  queens on an  $N \times N$  chessboard so that none of them are able to capture others. Write a function that calculates them all and prints them to the screen as a visual board layout, and returns the number of solutions it found.

**Tip:** A queen moves in straight lines vertically, horizontally, or diagonally. You need to place all eight queens so that no two share the same row, column, or diagonal.

**Tip:** In the more advanced version of this challenge you would be required to return only the fundamental solutions, which means unique positions excluding rotations and reflections. This is not a requirement here.

## Sample input and output

- In an 8x8 board you need to place 8 queens. There are 92 possible arrangements, so your function should print each of them then return 92.
- In a 10x10 board you need to place 10 queens. There are 724 possible arrangements, so your function should print each of them then return 724.

Here is a suggested example layout for solutions:

```
.....Q  
..Q....  
Q.....  
..Q.....  
.....Q..  
.Q.....  
.....Q.  
....Q...  
.....Q..
```

## Hints

**Hint #1:** Your queen placement function ought to call itself. The first time it's called it represents the first queen being placed, the second it's the second queen, and so on. If you reach  $N$  then you've placed all the queens and you have a solution.

**Hint #2:** In order to be sure you've found all solutions, you need to exhaust all possible placements.

**Hint #3:** Sometimes you will have placed six queens and realized there's nowhere valid for the seventh to go. Be prepared to backtrack.

**Hint #4:** Two queens occupy the same column if their X difference is equal to their Y difference, or their X difference is equal to their negative Y difference.

**Hint #5:** You can solve this problem using a one-dimensional array of integers.

## Solution

At its core this is a simple recursion challenge: you need to try placing every queen in every row and column, then try placing the second queen in every row and column that can't be attacked by the first queen, then try placing the third queen in every row and column that can't be attacked by the previous two queens, and so on.

Using  $N \times N$  is a sneaky trick, because it makes the challenge sound harder when really nothing really changes. Sure, you count to  $N$  rather than hard-coding 8, but that's it - nothing else changes.

I'm going to give you my solution shortly, as well as a full explanation so you can compare it to your own. However, first I want to discuss one part of the solution that might confuse you: I used a one-dimensional integer array. This means a solution for me looks like this:

[ 5, 2, 4, 7, 0, 3, 1, 6 ]

## Algorithms

Each element represents one column in the table, and its position in the array represents the row. So, element 0 is 5, which means row 0 column 5 contains a queen. Each queen is placed in its equivalent row number, which also means that each queen is placed in the array position that matches its queen number. So, the first queen (queen 0) is also placed at element 0, and in the solution above occupies the six column (column 5, counting from 0.)

Using this approach, consider these three board positions:

```
[0, 0, 0, 0, 0, 0, 0, 0]  
[7, 7, 7, 7, 7, 7, 7, 7]  
[0, 1, 2, 3, 4, 5, 6, 7]
```

Hopefully you can see that the first one represents eight queens lined up in the far-left column, the second one represents eight queens line up in the far-right column, and the third one represents the eight queens lined up diagonally top left to bottom right.

Now, when my array starts I pre-fill it with zeros, so it starts life looking like this:

```
[0, 0, 0, 0, 0, 0, 0, 0]
```

When I need to place a new queen into that array, I need to check it can't be attacked by any existing queen. This is done using two loops:

```
boardLoop: for column in 0 ..< board.count {  
    for row in 0 ..< queenNumber {
```

The first loop contains a **boardLoop** label so that we can jump to the next column if we find any queen in the current column while inside the inner loop. The second loop counts from 0 up to our current queen number, which is important: because we place queens in order (0 first, 1 second, etc) and because queens use their number for their row (queen 0 is always on row 0, queen 1 is always on row 1), we don't check beyond our current row.

This is both more efficient and required: the default value of our array is [0, 0, 0, 0, 0, 0, 0, 0],

so if we examined the full board it would look like eight rows of queen 0 at column 0 – i.e., no move is valid. By counting from 0 to less than the queen number, we only consider rows before our current one, and so never hit this problem.

OK, enough discussion, let's talk about the whole structure of my solution:

1. Create a one-dimensional array of eight integers, all initialized to zero, to represent the board.
2. Call my **challenge64()** function with the board, asking it to place queen 0.
3. If the queen number is equal to the size of the board, it means we've hit a solution so we can print it out. In our case, though, we're placing queen 0 so that won't trigger.
4. Enter two loops, as discussed above: the outer loop over every column, and the inner loop over the first  $N$  rows, where  $N$  is our queen number. We are placing queen 0, so this won't be used just yet.
5. If we were placing a queen higher than 0, the loop checks that the row and column in question can't be attacked by previously placed queens. If any queen occupies the row and column being checked, we exit the inner loop and jump to the next column.
6. If we find a row and column with a valid move we take it, then recursively call ourselves. This time, though, we add 1 to the queen number being placed, so that we're placing the next queen.

One small extra complexity is the need to return the total number of solutions that are found. To make that work, my function returns an integer representing the number of solutions it found. If it hits a solution it returns 1, otherwise it sums the return values from its recursive calls.

So, that's how it works. Time for the code, complete with lots of comments to make sure it's clear:

```
func challenge64(board: [Int], queen queenNumber: Int) -> Int {
    if queenNumber == board.count {
        // we hit a solution – print it out
        print("Solution:", board)
```

## Algorithms

```
for row in 0 ..< board.count {
    for col in 0 ..< board.count {
        if board[row] == col {
            print("Q", terminator: "")
        } else {
            print(".", terminator: "")
        }
    }

    print("")
}

print("")
return 1
} else {
    // keep track how many solutions were found by our
    recursive calls
    var count = 0

    // loop over every column
    boardLoop: for column in 0 ..< board.count {

        // check only queens that are placed already
        for row in 0 ..< queenNumber {
            // find where this queen was placed
            let otherQueenColumn = board[row]

            // if this queen is placed in the column we are
            checking, stop checking other queens and go to the next column
            if otherQueenColumn == column { continue
        }

    }
}
```

```

        // calculate the difference in our row/column and
the checking queen's row/column
        let deltaRow = queenNumber - row
        let deltaCol = otherQueenColumn - column

        // if we are on a diagonal with this queen, stop
checking other queens and proceed to the next column
        if deltaRow == deltaCol { continue boardLoop }
        if deltaRow == -deltaCol { continue boardLoop }
    }

    // if we're still here it means this move is valid, so
take a copy of the board and make the move on the copy
    var boardCopy = board
    boardCopy[queenNumber] = column

    // now call ourselves recursively, placing one queen
number higher, and adding its return value to our solution
counter
    count += challenge64(board: boardCopy, queen:
queenNumber + 1)
}

// return our solution counter to the caller
return count
}

}

// create an initial board full of zeros
let emptyBoard = [Int](repeating: 0, count: 8)

```

## Algorithms

```
// call the solution function with the first queen and an empty
board
let solutionCount = challenge64(board: emptyBoard, queen: 0)
print("Found \(solutionCount) solutions")
```

Again, notice how the  $N \times N$  part of the problem made almost no difference – it just means looping up to **board.count** rather than 8.

With comments added my solution might look like a lot of code, but I hope you can see the algorithm itself is quite simple: it just brute-force places queens until it has placed them all.

Using a one-dimensional array for my board and placing queens by row helps reduce the workload, and even a modest Mac can find all 2,279,184 solutions in a 15x15 board in under a minute.

If you don't mind using **inout** for your board rather than making changes on a copy, you can make the algorithm perform more than 2x faster with no other changes. In my opinion, using **inout** for speed is only a good idea if you're working with performance-critical code – the principle of least astonishment applies to code too, and making copies of your data structure is easier to understand.



# Expert

## Question 1

What output will be produced by the code below?

```
func getNumber() -> Int {  
    print("Fetching number...")  
    return 5  
}  
  
func printStatement(_ result: @autoclosure () -> Bool) {  
    // print("Here is the number: \(result())")  
    print("Nothing to see here.")  
}  
  
printStatement(getNumber() == 5)
```

- A. "Fetching number"
- B. "Nothing to see here."
- C. Nothing will be output
- D. This code will compile but crash
- E. This code will not compile
- F. "Fetching number...", "Nothing to see here."

## Question 2

In the code below, what data type is **testVar**?

```
let names = ["Pilot": "Wash", "Doctor": "Simon"]  
  
for (key, value) in names.enumerated() {  
    let testVar = value
```

}

- A. [String, String]
- B. String
- C. This code will not compile
- D. (String, String)

### Question 3

What output will be produced by the code below?

```
let status = "shiny"

for (position, character) in status.reversed().enumerated()
where position % 2 == 0 {
    print("\(position): \(character)")
}
```

- A. Nothing will be output
- B. 0: y, 2: i, 4: s
- C. This code will compile but crash
- D. This code will not compile
- E. 0: s, 2: i, 4: y
- F. 0: s
- G. 0: y
- H. 4: y, 2: i, 0: s
- I. 4: s, 2: i, 0: y

### Question 4

What output will be produced by the code below?

```
var spaceships1 = Set<String>()
spaceships1.insert("Serenity")
spaceships1.insert("Enterprise")
spaceships1.insert("TARDIS")

let spaceships2 = spaceships1

if spaceships1.isSubset(of: spaceships2) {
    print("This is a subset")
} else {
    print("This is not a subset")
}
```

- A. This code will compile but crash
- B. "This is not a subset"
- C. This code will not compile
- D. "This is a subset"
- E. Nothing will be output

## Question 5

In the code below, what is the data type used for **result**?

```
func doStuff() {
    print("Stuff has been done")
}
```

```
let result = doStuff()
```

- A. This code will not compile

- B. () (a tuple)
- C. 0
- D. [] (an array)
- E. nil
- F. false

## Question 6

What output will be produced by the code below?

```
func greet(var name: String) {  
    name = name.uppercased()  
    print("Greetings, \(name)!")  
}  
  
greet("River")
```

- A. This code will not compile
- B. "Greetings, River!"
- C. "River"
- D. Nothing will be output
- E. This code will compile but crash
- F. "GREETINGS, RIVER!"

## Question 7

What output will be produced by the code below?

```
func foo(_ number: Int) -> Int {  
    func bar(_ number: Int) -> Int {  
        return number * 5  
    }  
    return bar(number)  
}
```

```
    }

    return number * bar(3)
}

print(foo(2))
```

- A. Nothing will be output
- B. 15
- C. 5
- D. This code will not compile
- E. 2
- F. 10
- G. 3
- H. 30
- I. This code will compile but crash

## Question 8

What output will be produced by the code below?

```
let names = ["River", "Kaylee", "Zoe"]
let result = names.sorted { $0 > $1 }
print(result)
```

- A. ["Kaylee"]
- B. This code will compile but crash
- C. ["Kaylee", "River", "Zoe"]
- D. ["Zoe", "River", "Kaylee"]
- E. This code will not compile
- F. Nothing will be output
- G. ["Zoe"]

## Question 9

Given the code below, what data type will be assigned to **test**?

```
enum Planet: Int {  
    case Mercury = 1  
    case Venus  
    case Earth  
    case Mars  
}  
  
let test = Planet(rawValue: 3)
```

- A. Planet?
- B. This code will compile but crash
- C. Planet
- D. This code will not compile
- E. Int
- F. Int?

## Question 10

When this code is executed, what will the **name** value contain?

```
let i = 11  
let j = 20  
  
let name = 9...11 ~= i ? j % 2 == 1 ? "Larry" : "Laura" :  
"Libby"
```

- A. nil
- B. "Laura"
- C. This code will compile but crash
- D. "Libby"
- E. This code will not compile
- F. "Larry"
- G. ""

## Question 11

What output will be produced by the code below?

```
let i = 101

if case 100...101 = i {
    print("Hello, world!")
} else {
    print("Goodbye, world!")
}
```

- A. "Goodbye, world!"
- B. Nothing will be output
- C. "Hello, world!"
- D. This code will not compile
- E. This code will compile but crash

## Question 12

What output will be produced by the code below?

```
let names: [String?] = ["Barbara", nil, "Janet", nil, "Peter",
```

```
nil, "George"]  
  
if let firstName = names.first {  
    print(firstName)  
}
```

- A. Optional("Barbara")
- B. nil
- C. This code will not compile
- D. This code will compile but crash
- E. Nothing will be output
- F. "Barbara"

### Question 13

What output will be produced by the code below?

```
func greet(_ name: String = "Anonymous") {  
    print("Hello, \(name)!")  
}  
  
let greetCopy = greet  
greetCopy("Dave")
```

- A. "Dave"
- B. This code will compile but crash
- C. Nothing will be output
- D. "Hello, Dave!"
- E. "Hello, "
- F. This code will not compile

## Question 14

Once this code is executed, what value will **result** hold?

```
let numbers = [1, 3, 5, 7, 9]
let result = numbers.reduce(0, +)
```

- A. (1, 3, 5, 7, 9) (a tuple)
- B. "1, 3, 5, 7, 9" (a string)
- C. 1
- D. "13579" (a string)
- E. [1, 3, 5, 7, 9] (an array)
- F. 25

## Question 15

Once this code is executed, how many items will be in the **result** array?

```
let names: [String?] = ["Barbara", nil, "Janet", nil, "Peter",
nil, "George"]
let result = names.flatMap { $0 }
```

- A. 7
- B. This code will compile but crash
- C. 3
- D. 0
- E. This code will not compile
- F. 4

## Question 16

What output will be produced by the code below?

```
var first = [1, 2, 3]
var second = ["one", "two", "three"]
var third = Array(zip(first, second))
print(third)
```

- A. [1, "one", 2, "two", 3, "three"]
- B. This code will compile but crash
- C. This code will not compile
- D. [(1, "one"), (2, "two"), (3, "three")]
- E. [(1), ("one"), (2), ("two"), (3), ("three")]
- F. (1, "one", 2, "two", 3, "three")
- G. [[1, "one"], [2, "two"], [3, "three"]]
- H. Nothing will be output

## Question 17

What output will be produced by the code below?

```
struct Spaceship {
    fileprivate(set) var name = "Serenity"
}

var serenity = Spaceship()
serenity.name = "Jayne's Chariot"
print(serenity.name)
```

- A. "Jayne's Chariot"
- B. Nothing will be output
- C. This code will not compile

- D. This code will compile but crash
- E. nil
- F. "Serenity"
- G. ""

## Question 18

What output will be produced by the code below?

```
let string: String = String(describing: String.self)  
print(string)
```

- A. This code will not compile
- B. Swift.String
- C. nil
- D. This code will compile but crash
- E. Nothing will be output
- F. ""
- G. "String"

## Question 19

What output will be produced by the code below?

```
let (captain, engineer, doctor) = ("Mal", "Kailee", "Simon")  
print(engineer)
```

- A. This code will not compile
- B. "Kailee"
- C. Nothing will be output
- D. ("Mal", "Kailee", "Simon") (a tuple)

- E. ("Kailee") (a tuple)
- F. "Simon"
- G. This code will compile but crash
- H. "Mal"
- I. ["Mal", "Kailee", "Simon"] (an array)

## Question 20

What output will be produced by the code below?

```
func square<T>(_ value: T) -> T {  
    return value * value  
}  
  
print(square(5))
```

- A. 5
- B. 25
- C. This code will not compile
- D. Nothing will be output
- E. This code will compile but crash

## Question 21

What output will be produced by the code below?

```
struct TaylorFan {  
    static var favoriteSong = "Shake it Off"  
    var name: String  
    var age: Int  
}
```

```
let fan = TaylorFan(name: "James", age: 25)
print(fan.favoriteSong)
```

- A. nil
- B. This code will compile but crash
- C. Nothing will be output
- D. "Shake it Off"
- E. This code will not compile

## Question 22

What output will be produced by the code below?

```
struct TaylorFan {
    fileprivate var name: String
}

var fan = TaylorFan(name: "Kailee")
fan.name = "Simon"
print(fan.name)
```

- A. Nothing will be output
- B. This code will not compile
- C. "Simon"
- D. "Kailee"
- E. ""
- F. This code will compile but crash
- G. nil

## Question 23

What output will be produced by the code below?

```
func foo(_ function: (Int) -> Int) -> Int {
    return function(function(5))
}

func bar<T: BinaryInteger>(_ number: T) -> T {
    return number * 3
}

print(foo(bar))
```

- A. 45
- B. 5
- C. 15
- D. 3
- E. Nothing will be output
- F. This code will not compile
- G. This code will compile but crash

## Question 24

Please read the code below:

```
import Foundation
let data: [Any?] = ["Bill", nil, 69, "Ted"]

for datum in data where datum is String? {
    print(datum)
}
```

```
for case let .some(datum) in data where datum is String {  
    print(datum)  
}
```

As you can see, there are two **for** loops, both operating on the **data** array. Which loop prints the most lines?

- A. The first loop prints more lines than the second
- B. The second loop prints more lines than the first
- C. This code will not compile
- D. This code will compile but crash
- E. Both loops print the same number of lines

## Question 25

How many lines will be printed by the code below?

```
import Foundation  
  
let data: [Any?] = ["Bill", nil, 69, "Ted"]  
  
for datum in data where !(datum is Hashable) {  
    print(datum)  
}
```

- A. No lines will be printed
- B. 1
- C. This code will not compile
- D. This code will compile but crash
- E. 2
- F. 3

- G. 4
- H. 0

## Question 26

In the code below, what data type will be assigned to **result**?

```
let result = UInt8.max.addingReportingOverflow(1)
```

- A. This code will not compile
- B. Array
- C. Tuple
- D. This code will compile but crash
- E. UInt8
- F. UInt
- G. Int
- H. Int8

## Question 27

When this code is executed, what will **result** be set to?

```
func fetchCrewMember() -> (job: String, name: String) {  
    return ("Public Relations", "Jayne")  
}
```

```
let result = fetchCrewMember().0
```

- A. "Public Relations"
- B. (0) (a tuple)
- C. This code will not compile

- D. "Jayne"
- E. ["Public Relations", "Jayne"] (an array)
- F. This code will compile but crash
- G. ("Public Relations", "Jayne") (a tuple)

## Question 28

What output will be produced by the code below?

```
var foo = bar = 0
bar *= 5
print(foo)
```

- A. 5
- B. 0
- C. 1
- D. Nothing will be output
- E. 00000
- F. This code will compile but crash
- G. This code will not compile

## Question 29

When this code is executed, what will **example2** be set to?

```
var names = [String]()
names.append("Amy")
let example1 = names.removeLast()
let example2 = names.removeLast()
```

- A. "Amy"

- B. Nothing will be output
- C. This code will compile but crash
- D. ""
- E. This code will not compile
- F. nil

## Question 30

When this code is executed, what will **example2** be set to?

```
var names = [String]()
names.append("Amy")

let example1 = names.popLast()
let example2 = names.popLast()
```

- A. nil
- B. This code will compile but crash
- C. Nothing will be output
- D. "Amy"
- E. ""
- F. This code will not compile

## Answers (Expert)

**Question 1:** The correct answer is **B**, 'Nothing to see here'. The **@autoclosure** attribute automatically turns the code **getNumber() == 5** into a closure that would have been called in the "Here is the number line" had it not been commented out. Because that closure is never used, the "Fetching number..." message is never printed, leaving just "Nothing to see here."

being displayed.

**Question 2:** The correct answer is **D**, '(String, String)'. If you iterate over the **names** dictionary without using **enumerated()**, **key** would be Pilot then Doctor, and **value** would be Wash then Simon. However, this code uses **enumerated()**, which means that **key** will be the integer position of the item in the loop, and **value** will be a **(String, String)** tuple containing the key and the value for this item in the dictionary: ("Pilot", "Wash") then ("Doctor", "Simon").

**Question 3:** The correct answer is **B**, '0: y, 2: i, 4: s'. There are three things to understand about this code. First, **reversed()** is called on the string before **enumerated()**, which means the string is reversed but the enumeration (the positions) are not. Second, **enumerated()** will return each character in the reversed string, along with its position. Third, that position is passed into a **where** clause, and only even-numbered character indices will be printed.

**Question 4:** The correct answer is **D**, 'This is a subset'. Sets distinguish between subsets and strict (or "proper") subsets, with the difference being that the latter necessarily excludes identical sets. That is, Set A is a subset of Set B if every item in Set A is also in Set B. On the other hand, Set A is a strict subset of Set B if every element in Set A is also in Set B, but Set B contains at least one item that is missing from Set A.

**Question 5:** The correct answer is **B**, '()' (a tuple). The **doStuff()** function is not declared as returning a value, so Swift will automatically consider it to return **Void**, which itself is a typealias for the empty tuple type, **()**.

**Question 6:** The correct answer is **A**, 'This code will not compile'. The **greet()** method declares its **name** parameter using the **var** keyword, which was allowed only in old versions of Swift. From Swift 3.0 onwards this construction is no longer valid, so the compiler will refuse to build this code.

**Question 7:** The correct answer is **H**, '30'. This code uses inner functions, which means that the **foo()** function is visible at the end whereas the **bar()** function is not. The final line calls **foo()** with a value of 2; this then multiplies that by the result of calling **bar()** with a value of 3, which in turn multiplies that 3 by 5. So, you get  $3 * 5 * 2$ , i.e. 30.

**Question 8:** The correct answer is **D**, '["Zoe", "River", "Kaylee"]'. The closure used by

`sorted()` returns true if the item in `$0` should be sorted before the item in `$1`. In this case, we're comparing using `>`, which means that items are sorted earlier when they are later in the alphabet, giving a reverse sort.

**Question 9:** The correct answer is **A**, 'Planet?'. Creating an enum from its raw value might fail: we could have used **99** for the raw value, and that clearly does not exist. So, this will return either a planet enum or **nil** if the planet was not found.

**Question 10:** The correct answer is **B**, 'Laura'. This code uses the ternary operator (`?:`) and the pattern match operator (`~=`) to choose one of three names. First, the pattern match operator is used to decide whether the value of `i` (11) is in the range 9...11, which is just the numbers 9, 10, and 11. If the pattern match operator returns true, a second check happens: is the value of `j` (20) divisible by two with a remainder of one? If so, the name Larry is used; if not, Laura. And if the pattern match operator returns false, Libby is used. In this case, the pattern match succeeds but 20 divided by 2 does not leave a remainder of 1, so Laura is used for **name**.

**Question 11:** The correct answer is **C**, 'Hello, world!'. Using **if case** with a range like **100...101** allows us to check whether an integer is inside that range. In this case, it checks whether the value of `i` (101) is inside the range of numbers 100 and 101 (inclusive), which it is, so "Hello, world!" is printed.

**Question 12:** The correct answer is **A**, 'Optional("Barbara")'. The **names** array contains values of type **String?**, but **names.first** adds an extra level of optionality because it will return **nil** if there are no items in the array. So, in this code **names.first** will return **String??** (an optional optional String), of which one layer is unwrapped using the **if let**.

**Question 13:** The correct answer is **D**, 'Hello, Dave!'. This code assigns the **greet()** method to the **greetCopy** constant, which means you can then call **greetCopy()** as if it were **greet()**.

**Question 14:** The correct answer is **F**, '25'. The **reduce** method combines the integers in the **numbers** array into a single value by applying a function to each one. In our case, that's the `+` operator, which means "add all these numbers together", giving 25.

**Question 15:** The correct answer is **F**, '4'. Swift's **flatMap** will automatically strip nil values from an array, meaning that **result** will contain the names Barbara, Janet, Peter, and George.

**Question 16:** The correct answer is **D**, `[(1, "one"), (2, "two"), (3, "three")]`. This code creates a **Zip2Sequence** struct out of two arrays: one containing 1, 2, 3, and another containing "one", "two", "three". The **Zip2Sequence** will match each item in the first array with the item at the same index in the second array, giving (1, "one") and so on. This is then converted to an array for easier access.

**Question 17:** The correct answer is **A**, 'Jayne's Chariot'. Creating a property using **fileprivate(set)** means that Swift won't let code from other files modify that property directly. However, Swift playgrounds are compiled into a single file when run, and code that exists in the same file as a **fileprivate** property can access it directly.

**Question 18:** The correct answer is **G**, 'String'. Among the many constructors for strings is one that lets you pass in a class to have the string set to the name of that class. That is, **String(describing: String.self)** means "create a string out of the name of the String class." This is equivalent to the **NSStringFromClass()** function that Objective-C developers often use.

**Question 19:** The correct answer is **B**, 'Kailee'. This code demonstrates tuple destructuring, which is a method of pulling a tuple into multiple individual values in one line of code. In this example, it will create three constants (**captain**, **engineer**, and **doctor**), giving them the values "Mal", "Kailee", and "Simon" respectively.

**Question 20:** The correct answer is **C**, 'This code will not compile'. This code has attempted to create a generic number squaring function, but has failed to declare that **T** (the data type being used) has the ability to work with the \* operator, so Swift cannot compile it. One solution might be to modify it to **square<T: BinaryInteger>**, which would allow it to work on **Int**, **UInt**, **Int64** and so on.

**Question 21:** The correct answer is **E**, 'This code will not compile'. The **favoriteSong** property has been declared **static**, which means it must be accessed using **TaylorFan.favoriteSong**.

**Question 22:** The correct answer is **C**, 'Simon'. Creating a property using **fileprivate** means that Swift won't let code from other files read or write that property directly. However, Swift playgrounds are compiled into a single file when run, and code that exists in the same file as a **fileprivate** property can access it directly.

**Question 23:** The correct answer is **A**, '45'. This code combines generics and closures to make something that's easy to read but hard to understand. It passes the **bar()** function into **foo()** as its only parameter. That **bar()** method is then called twice: once with 5 as its parameter, and then again with the result from the first call as its parameter. The first call passes in 5, so **bar()** returns  $5 * 3$ , i.e. 15. The second call passes in that 15, so **bar()** returns  $15 * 3$ , i.e. 45.

**Question 24:** The correct answer is **A**, 'The first loop prints more lines than the second'. There is a very subtle difference between the two loops, and it's triggered by the data type of the array: this is an array of **Any?** not an array of strings. The first loop will attempt to typecast its items as **String?**, which means the loop element must either be a string or nil – that's true of three items. The second loop, however, begins by unwrapping the optional, so it will either be **Any** or **String**, at which point our **where** clause will work. So, the second loop prints two lines.

**Question 25:** The correct answer is **B**, '1'. The loop will print only items that do not conform to the hashable protocol. In this code, the only such item is **nil**, so one line will be printed.

**Question 26:** The correct answer is **C**, 'Tuple'. The **UInt8.addingReportingOverflow()** method returns a tuple (**UInt8, Bool**) where the first value stores the result of the add operation, and the second value stores whether an overflow happened.

**Question 27:** The correct answer is **A**, 'Public Relations'. Even though we name the elements in **fetchCrewMember()**'s tuple return value, those elements are still accessible using their numerical index as used in the code.

**Question 28:** The correct answer is **G**, 'This code will not compile'. Many programming languages allow syntax like **foo = bar = 0**, but Swift is not one of them: this code will not compile.

**Question 29:** The correct answer is **C**, 'This code will compile but crash'. The **removeLast()** method returns the same data type as the array contains, which in this code is a String. As the only string in the array was already removed, the second call will throw an exception and crash.

**Question 30:** The correct answer is **A**, 'nil'. The **popLast()** method returns an optional version

of the data type the array contains, which in this code is a **String?**. As the only string in the array was already removed, the second call will return **nil**.

# Novice

## Question 1

What output will be produced by the code below?

```
let i = 3

switch i {
    case 1:
        print("Number was 1")
    case 2:
        print("Number was 2")
    case 3:
        print("Number was 3")
}
```

- A. "Number was 3"
- B. "Number was 1"
- C. Nothing will be output
- D. "Number was 2"
- E. This code will compile but crash
- F. This code will not compile

## Question 2

Given the code below, what data type does **i** have?

```
let i = 10.2
```

- A. This code will not compile
- B. Int
- C. Decimal
- D. Single
- E. This code will compile but crash
- F. Double
- G. Float

### Question 3

When this code is executed, what is the value of **myStr**?

```
var myStr: String = "0"  
myStr = "shiny"
```

- A. "shiny"
- B. nil
- C. This code will compile but crash
- D. "nil"
- E. ""
- F. This code will not compile

### Question 4

When this code is executed, what is the value of the **swift** string?

```
import Foundation  
let swift = "Hello"  
let ns = NSString(swift)
```

- A. This code will not compile
- B. ""
- C. nil
- D. "Hello"
- E. This code will compile but crash

## Question 5

What output will be produced by the code below?

```
let names = [ "Amy" , "Rory" ]  
  
for name in names {  
    name = name.uppercased()  
    print( "HELLO, \(name)!" )  
}
```

- A. Nothing will be output
- B. This code will not compile
- C. "HELLO, AMY!"
- D. "HELLO, AMY!", "HELLO, RORY!"
- E. "HELLO, RORY!"
- F. This code will compile but crash

## Question 6

How many bits are used to store an **Int**?

- A. 64-bit
- B. It depends on the device
- C. 32-bit

D. 16-bit

## Question 7

What output will be produced by the code below?

```
let names = [ "Amy" , "Clara" ]  
  
for i in 0 ... names.length {  
    print("Hello, \(names[i])!")  
}
```

- A. This code will compile but crash
- B. "Hello, Amy!", "Hello, Clara!"
- C. "Hello, Amy!"
- D. "Hello, Clara!"
- E. This code will not compile

## Question 8

When this code finishes executing, how many strings will the **names** array contain?

```
let names = [String]()  
names.append("Amy")  
names.append("Clara")  
names.append("Rory")
```

- A. 1
- B. 2
- C. 3
- D. This code will not compile

- E. 0
- F. This code will compile but crash

## Question 9

What output will be produced by the code below?

```
let names = ["Chris", "Joe", "Doug", "Jordan"]  
  
if let name = names[1] {  
    print("Brought to you by \(name)")  
}
```

- A. This code will compile but crash
- B. "Brought to you by Joe"
- C. This code will not compile
- D. "Brought to you by Chris"

## Question 10

What output will be produced by the code below?

```
var i = 2  
  
do {  
    print(i)  
    i *= 2  
} while (i < 128)
```

- A. 2, 4, 8, 16, 32, 64
- B. 4, 8, 16, 32, 64

- C. 2, 4, 8, 16, 32, 64, 128
- D. 4, 8, 16, 32, 64, 128
- E. This code will not compile
- F. This code will compile but crash

## Question 11

What output will be produced by the code below?

```
for i in 1...3 {  
    print(i)  
}
```

- A. 1, 2, 3
- B. 2, 3
- C. 2
- D. This code will compile but crash
- E. This code will not compile
- F. Nothing will be output
- G. 1, 2

## Question 12

What output will be produced by the code below?

```
for i in 3...1 {  
    print(i)  
}
```

- A. 2
- B. 1, 2

- C. Nothing will be output
- D. This code will compile but crash
- E. 3, 2
- F. 1, 2, 3
- G. 3, 2, 1
- H. This code will not compile

### Question 13

What output will be produced by the code below?

```
class Starship {  
    var type: String  
    var age: Int  
}
```

```
let serenity = Starship(type: "Firefly", age: 24)  
print(serenity.type)
```

- A. Nothing will be output
- B. ""
- C. This code will compile but crash
- D. This code will not compile
- E. "Serenity"
- F. nil
- G. "Firefly"

### Question 14

What output will be produced by the code below?

```

struct Starship {
    var name: String
}

let tardis = Starship(name: "TARDIS")
var enterprise = tardis
enterprise.name = "Enterprise"

print(tardis.name)

```

- A. nil
- B. "TARDIS"
- C. "Enterprise"
- D. This code will compile but crash
- E. Nothing will be output
- F. This code will not compile
- G. ""

## Question 15

What output will be produced by the code below?

```

import Foundation
let number = 16.0
print("\(number) squared is \(number * number), and its square
root is \(sqrt(number))")

```

- A. This code will compile but crash
- B. "16 squared is 256, and its square root is 4"
- C. "16 squared is 16 \* 16, and its square root is sqrt(16)"
- D. "16.0 squared is 256.0, and its square root is 4.0"
- E. This code will not compile

## Question 16

What output will be produced by the code below?

```
func sayHello(to name: String) -> String {  
    return "Howdy, \(name)!"  
}  
  
print("\(sayHello(to: "Jayne"))")
```

- A. "Howdy!"
- B. Nothing will be output
- C. "sayHello(to: "Jayne")"
- D. This code will compile but crash
- E. "Howdy, Jayne!"
- F. This code will not compile

## Question 17

What output will be produced by the code below?

```
struct Spaceship {  
    var name: String {  
        willSet {  
            print("I'm called \(newValue)!")  
        }  
    }  
}  
  
var serenity = Spaceship(name: "Serenity")
```

```
serenity.name = "TARDIS"
```

- A. "I'm called TARDIS!"
- B. This code will not compile
- C. "I'm called Serenity!"
- D. This code will compile but crash
- E. "I'm called Serenity!", "I'm called TARDIS!"
- F. Nothing will be output

## Question 18

What output will be produced by the code below?

```
enum Weather {  
    case sunny  
    case cloudy  
    case windy(speed: Int)  
}  
  
let today: Weather = .windy(speed: 10)  
  
switch today {  
case .sunny, .cloudy:  
    print("It's not that windy")  
case .windy(let speed) where speed >= 10:  
    print("It's very windy")  
default:  
    print("It's a bit windy")  
}
```

- A. This code will compile but crash
- B. "It's very windy"

- C. This code will not compile
- D. "It's not that windy"
- E. Nothing will be output
- F. "It's a bit windy"

## Question 19

What output will be produced by the code below?

```
let names = ["Serenity", "Sulaco", "Enterprise", "Galactica"]

for name in names where name.hasPrefix("S") {
    print(name)
}
```

- A. "Sulaco"
- B. This code will compile but crash
- C. This code will not compile
- D. "Serenity"
- E. "Serenity", "Sulaco"
- F. Nothing will be output
- G. "Serenity", "Sulaco", "Enterprise"

## Question 20

What output will be produced by the code below?

```
let oneMillion = 1_000_000
let oneThousand = oneMillion / 0_1_0_0_0
print(oneThousand)
```

- A. 1\_000\_000
- B. This code will compile but crash
- C. 1000
- D. 1000000
- E. 1\_0\_0\_0
- F. This code will not compile
- G. 1\_0\_0\_0\_0\_0\_0
- H. Nothing will be output

## Question 21

What output will be produced by the code below?

```
let names = ["Serenity", "Sulaco", "Enterprise", "Galactica"]

if let last = names.last {
    print(last)
}
```

- A. Nothing will be output
- B. Optional("Galactica")
- C. "Serenity"
- D. Optional("Serenity")
- E. This code will not compile
- F. This code will compile but crash
- G. nil
- H. "Galactica"

## Question 22

You are using a method named **willThrowAnError()**, and it is marked using **throws** because it will always throw the error "Forbidden". Given this context, what output will be produced by the code below?

```
do {  
    try willThrowAnError()  
} catch {  
    print("The error message was: \(error)")  
}
```

- A. This code will compile but crash
- B. "The error message was: Forbidden"
- C. Nothing will be output
- D. This code will not compile

### Question 23

What output will be produced by the code below?

```
var motto = "Bow ties are cool"  
motto.replacingOccurrences(of: "Bow", with: "Neck")  
print(motto)
```

- A. "Neck ties are cool"
- B. Nothing will be output
- C. "ties are cool"
- D. This code will not compile
- E. "Bow ties are cool"
- F. "Bow"
- G. This code will compile but crash

## Question 24

When this code is executed, what will the **third** constant contain?

```
let first = ["Sulaco", "Nostromo"]
let second = ["X-Wing", "TIE Fighter"]
let third = first + second
```

- A. It will be an empty array
- B. This code will compile but crash
- C. "Sulaco", "Nostromo", "X-Wing", "TIE Fighter"
- D. This code will not compile
- E. "Sulaco", "Nostromo"
- F. "Sulaco", "Nostromo", "Sulaco", "Nostromo"
- G. "X-Wing", "TIE Fighter", "Sulaco", "Nostromo"

## Question 25

What output will be produced by the code below?

```
final class Dog {
    func bark() {
        print("Woof!")
    }
}

class Corgi : Dog {
    override func bark() {
        print("Yip!")
    }
}
```

```
let muttface = Corgi()  
muttface.bark()
```

- A. "Yip!"
- B. Nothing will be output
- C. "Woof!"
- D. This code will compile but crash
- E. "Woof!", "Yip!"
- F. This code will not compile
- G. "Yip!", "Woof!"

## Question 26

When this code is executed, what will be the value of the **j** constant?

```
let i = "5"  
let j = i + i
```

- A. 5, 5 (an array of integers)
- B. This code will not compile
- C. This code will compile but crash
- D. "55" (a string)
- E. "5", "5" (an array of strings)
- F. 5 (an integer)
- G. "5" (a string)
- H. 55 (an integer)

## Question 27

What output will be produced by the code below?

```
var i = 2

repeat {
    i *= i * 2
} while i < 100

print(i)
```

- A. 2
- B. This code will not compile
- C. This code will compile but crash
- D. Nothing will be output
- E. This is an infinite loop
- F. 64
- G. 128

## Question 28

Once this code is executed, how many items will **numbers** contain?

```
var numbers = [1, 2, 3]
numbers += [4]
```

- A. This code will not compile
- B. This code will compile but crash
- C. 4
- D. 2
- E. 3
- F. It will be empty
- G. 1

## Question 29

When this code is executed, what value will **num** have?

```
let num = UInt.min
```

- A. 255
- B. -255
- C. 0
- D. This code will compile but crash
- E. 16,777,216
- F. This code will not compile
- G. -16,777,216

## Question 30

What output will be produced by the code below?

```
import Foundation
let number = 16
print("\(number) squared is \(number * number), and its square
root is \(sqrt(number))")
```

- A. 16 squared is 256, and its square root is 4
- B. This code will compile but crash
- C. This code will not compile
- D. 16 squared is 16 \* 16, and its square root is sqrt(16)
- E. 16.0 squared is 256.0, and its square root is 4.0

## Answers (Novice)

**Question 1:** The correct answer is **F**, 'This code will not compile'. Swift requires all switch statements to be exhaustive. This code will not compile because it does not have a **default** clause.

**Question 2:** The correct answer is **F**, 'Double'. When given a floating-point number, Swift's type inference will use the **Double** data type.

**Question 3:** The correct answer is **A**, 'shiny'. Swift constants do not have to be given an initial value, as long as they are given a value only once and before they are used.

**Question 4:** The correct answer is **A**, 'This code will not compile'. While it is possible to typecast an **NSString** into a **String** using an initializer without a label, it is not possible the other way around. The code should read **let ns = NSString(string: "Hello")**.

**Question 5:** The correct answer is **B**, 'This code will not compile'. This code will not compile because it modifies **name** inside the loop. If you want to do this, you must use the **var** keyword like this: **for var name in names**.

**Question 6:** The correct answer is **B**, 'It depends on the device'. Swift's integers are 32-bit on 32-bit devices such as iPhone 5 and earlier, and 64-bit on 64-bit devices such as iPhone 5s and later.

**Question 7:** The correct answer is **E**, 'This code will not compile'. Swift arrays do not have a **length** property; this code should use **count**. With that replacement the code will compile, however it will then crash because it uses the closed range operator (...), which will cause an out-of-bounds error when reading into the array.

**Question 8:** The correct answer is **D**, 'This code will not compile'. The **names** array was declared using **let**, which makes it a constant. This means it is a compile error to try to use **append()** to add strings to it.

**Question 9:** The correct answer is **C**, 'This code will not compile'. Subscripting an array of strings will return a **String** rather than a **String?**, which means it is a compile error to attempt to unwrap it using **if let**.

**Question 10:** The correct answer is E, 'This code will not compile'. The **do** keyword is invalid here; the programmer should use **repeat** instead.

**Question 11:** The correct answer is A, '1, 2, 3'. This uses the closed range operator (...) to loop from 1 to 3 inclusive.

**Question 12:** The correct answer is D, 'This code will compile but crash'. This code will be compiled successfully, but crash at runtime: Swift does not allow you to generate ranges where the initial value is greater than the end value.

**Question 13:** The correct answer is D, 'This code will not compile'. Structs have memberwise initialization as standard, but this is not available to classes. The code will fail to compile because the **Starship** class has no initializers.

**Question 14:** The correct answer is B, 'TARDIS'. Although **enterprise** was created as a copy of **tardis**, it is a struct and therefore a value type: changing **enterprise** will not also change **tardis**.

**Question 15:** The correct answer is D, '16.0 squared is 256.0, and its square root is 4.0'. Using its type inference, Swift will consider **number** to be an **Double**, which will be interpolated correctly into the string even when **sqrt()** is called. When **Doubles** are interpolated into strings, they have .0 appended to their values when they have no fractional digits.

**Question 16:** The correct answer is E, 'Howdy, Jayne!'. This is a basic example of string interpolation in action. As well as replacing simple values, Swift can also call a function as part of interpolation, as seen here.

**Question 17:** The correct answer is A, 'I'm called TARDIS!'. The **willSet** property observer is triggered only when the initial value is changed, and not when the struct is created using memberwise initialization.

**Question 18:** The correct answer is B, 'It's very windy'. The **windy** case value has an associated value to store the wind speed as an integer. In the code, this is set to 10, which means the "It's very windy" case will be triggered in the **switch** block.

**Question 19:** The correct answer is E, '"Serenity", "Sulaco"'. The **where** condition for the loop

will ensure that only names that start with the letter S will be used inside the loop.

**Question 20:** The correct answer is C, '1000'. Swift allows you to use any number of leading zeroes before a number, and any number of underscores inside a number, in order to make reading easier. The example given, `0_1_0_0_0`, is unlikely, but a perfectly valid way to write 1000.

**Question 21:** The correct answer is H, 'Galactica'. Using `names.last` will return an optional string, but the `if let` unwraps that safely to produce "Galactica".

**Question 22:** The correct answer is B, 'The error message was: Forbidden'. When in a `catch` block with no pattern, Swift automatically matches any error and binds it to a local constant called `error`.

**Question 23:** The correct answer is E, 'Bow ties are cool'. The `replacingOccurrences()` method returns a new string with its change effected, rather than modifying it in place. In this code, that return value is being discarded, so the original `motto` variable remains unchanged.

**Question 24:** The correct answer is C, '"Sulaco", "Nostromo", "X-Wing", "TIE Fighter"'. Swift arrays can be joined together using the `+` operator, which adds the right-hand array to the end of the left-hand.

**Question 25:** The correct answer is F, 'This code will not compile'. This code attempts to create a new class, `Corgi`, that inherits from an existing class, `Dog`. Ordinarily that would be fine, but here the `Dog` class has been marked as `final`, which means it cannot be inherited from.

**Question 26:** The correct answer is D, '55' (a string). When used with strings, the `+` operator acts to append one string to another. In this case, it merges "5" onto "5" to make "55".

**Question 27:** The correct answer is G, '128'. Each time the loop goes around, the `i` is doubled then multiplied by itself. The first time through the loop it will be 8, and the second time it will be 128, at which point the loop will exit and print 128.

**Question 28:** The correct answer is C, '4'. The `+=` operator, when applied to array, serves to append one array to another in place. In this case, it appends an array containing one item (4) to the existing array of three items.

**Question 29:** The correct answer is C, '0'. **UInt** means "unsigned integer", which is a whole number that cannot be less than zero.

**Question 30:** The correct answer is C, 'This code will not compile'. The **sqrt()** function can be called using two different types of parameter, neither of which are integers. In this example, Swift's type inference will consider **number** to be a **BinaryInteger**, which cannot be used by **sqrt()** unless you typecast it.

# Intermediate

## Question 1

What output will be produced by the code below?

```
let names = ["Pilot": "Wash", "Doctor": "Simon"]
let doctor = names["doctor"] ?? "Bones"
print(doctor)
```

- A. This code will not compile
- B. "Doctor"
- C. This code will compile but crash
- D. "Wash"
- E. Nothing will be output
- F. nil
- G. ""
- H. "Simon"
- I. "Bones"

## Question 2

When this code is executed, what value does **result** contain?

```

func sum(numbers: Int...) -> Int {
    var result = 0

    for number in numbers {
        result += number
    }

    return result
}

let result = sum(numbers: [1, 2, 3, 4, 5])

```

- A. This code will compile but crash
- B. 15
- C. 1, 2, 3, 4, 5
- D. This code will not compile
- E. Nothing will be output
- F. 0

### Question 3

What output will be produced by the code below?

```

func greet(names: String...) {
    print("Criminal masterminds:", names.joined(separator: ","))
}

greet(names: "Malcolm", "Kaylee", "Zoe")

```

- A. "Criminal masterminds: Array<String>"
- B. Nothing will be output

- C. "Criminal masterminds: Malcolm, Kaylee, and Zoe"
- D. This code will not compile
- E. "Criminal masterminds: Malcolm, Kaylee, Zoe"
- F. This code will compile but crash

## Question 4

What output will be produced by the code below?

```
let names = ["Simon", "River", "Book"]

names.forEach {
    print($1)
}
```

- A. This code will not compile
- B. "Simon"
- C. ["Simon", "River", "Book"]
- D. This code will compile but crash
- E. "Book", "Simon", "River"
- F. Nothing will be output
- G. "Simon", "River", "Book"

## Question 5

What output will be produced by the code below?

```
import Foundation

let rounded: Int = round(10.5)
print("Rounding 10.5 makes \(rounded)")
```

- A. "Rounding 10.5 makes 10"
- B. Nothing will be output
- C. "Rounding 10.5 makes 10.5"
- D. This code will compile but crash
- E. "Rounding 10.5 makes 11"
- F. This code will not compile

## Question 6

What output will be produced by the code below?

```
func fizzbuzz(number: Int) -> String {  
    switch (number % 3 == 0, number % 5 == 0) {  
        case (true, false):  
            return "Fizz"  
        case (false, true):  
            return "Buzz"  
        case (true, true):  
            return "FizzBuzz"  
        default:  
            return String(number)  
    }  
}  
  
print(fizzbuzz(number: 15))
```

- A. "Fizz"
- B. Nothing will be output
- C. "15"
- D. "FizzBuzz"
- E. This code will compile but crash
- F. "Buzz"

G. This code will not compile

### Question 7

What output will be produced by the code below?

```
struct Spaceship {  
    var name: String  
  
    func setName(_ newName: String) {  
        name = newName  
    }  
}  
  
var enterprise = Spaceship(name: "Enterprise")  
enterprise.setName("Enterprise A")  
print(enterprise.name)
```

- A. ""
- B. "Enterprise A"
- C. "Enterprise"
- D. This code will not compile
- E. Nothing will be output
- F. This code will compile but crash

### Question 8

What output will be produced by the code below?

```
for i in stride(from: 1, to: 17, by: 4) {  
    print(i)  
}
```

- A. 1, 4, 8, 12, 16
- B. 1, 5, 9, 13, 17
- C. This code will compile but crash
- D. 4, 8, 12
- E. Nothing will be output
- F. 1, 5, 9, 13
- G. This code will not compile
- H. 4, 8, 12, 16

## Question 9

Read the code below:

```
var foo: Float = 32
var bar: Double = 32
```

Assuming no typecasting, which of the following statements is true?

- A. Either variable can be assigned to the other
- B. Neither variable can be assigned to the other
- C. You can assign "foo" to "bar"
- D. This code will not compile
- E. This code will compile but crash
- F. You can assign "bar" to "foo"

## Question 10

What output will be produced by the code below?

```
let foo = 0x10
```

```
print(foo)
```

- A. 15
- B. 10
- C. This code will compile but crash
- D. 16
- E. Nothing will be output
- F. 0x10
- G. 11
- H. This code will not compile

## Question 11

What output will be produced by the code below?

```
let bigNumber = Int.max  
let biggerNumber = bigNumber + 1  
print(biggerNumber)
```

- A. Nothing will be output
- B. 0
- C. This code will not compile
- D. 9223372036854775807
- E. This code will compile but crash
- F. -9223372036854775808

## Question 12

When this code is executed, what data type does **convertedNumber** have?

```
let possibleNumber = "1701"
```

```
let convertedNumber = Int(possibleNumber)
```

- A. This code will not compile
- B. String
- C. Int
- D. Int?

### Question 13

What output will be produced by the code below?

```
var spaceships = Set<String>()
spaceships.insert("Serenity")
spaceships.insert("Enterprise")
spaceships.insert("TARDIS")
spaceships.insert("Serenity")
print(spaceships.count)
```

- A. 3
- B. 4
- C. Nothing will be output
- D. 0
- E. This code will not compile
- F. 1
- G. This code will compile but crash

### Question 14

What output will be produced by the code below?

```
var crew = [ "Captain": "Malcolm", "Doctor": "Simon" ]
```

```
crew = [:]  
print(crew.count)
```

- A. This code will compile but crash
- B. 2
- C. 0
- D. 3
- E. Nothing will be output
- F. 1
- G. This code will not compile

## Question 15

What output will be produced by the code below?

```
import Foundation  
let crew = NSMutableDictionary()  
crew.setValue("Kryten", forKey: "Mechanoid")  
print(crew.count)
```

- A. nil
- B. This code will compile but crash
- C. 0
- D. Nothing will be output
- E. This code will not compile
- F. 1

## Question 16

What output will be produced by the code below?

```

let number = 5

switch number {
case 0..<5:
    print("First group")
case 5...10:
    print("Second group")
case 0...5:
    print("Third group")
default:
    print("Fourth group")
}

```

- A. This code will not compile
- B. "Fourth group"
- C. "First group"
- D. Nothing will be output
- E. "Second group"
- F. This code will compile but crash
- G. "Third group"

## Question 17

What output will be produced by the code below?

```

let point = (556, 0)
switch point {
case (let x, 0):
    print("X was \u2028(x)\u2029")
case (0, let y):
    print("Y was \u2028(y)\u2029")
case let (x, y):

```

```
    print("X was \(x) and Y was \(y)")  
}
```

- A. Nothing will be output
- B. X was 556
- C. Y was 0
- D. X was 556 and Y was 0
- E. This code will compile but crash
- F. This code will not compile

## Question 18

What output will be produced by the code below?

```
enum MyError: Error {  
    case broken  
    case busted  
    case badgered  
}  
  
func willThrowAnError() {  
    throw MyError.busted  
}  
  
do {  
    try willThrowAnError()  
} catch MyError.busted {  
    print("Code was busted!")  
} catch {  
    print("Code just didn't work")  
}
```

- A. Nothing will be output
- B. "Code was busted!"
- C. "Code just didn't work"
- D. This code will not compile
- E. This code will compile but crash

## Question 19

What output will be produced by the code below?

```
var i = 1

mainLoop: repeat {
    i += 2

    switch i % 2 {
        case 0:
            break mainLoop
        default:
            break
    }
} while true

print("Complete!")
```

- A. This code will compile but crash
- B. This code will not compile
- C. ""
- D. "Complete!"

## Question 20

What output will be produced by the code below?

```
let name = "Simon"

switch name {
    case "Simon":
        fallthrough

    case "Malcom", "Zoe", "Kaylee":
        print("Crew")

    default:
        print("Not crew")
}
```

- A. Nothing will be output
- B. "Crew"
- C. This code will compile but crash
- D. "Not crew"
- E. This code will not compile

## Question 21

What output will be produced by the code below?

```
let people = [String](repeating: "Malkovitch", count: 2)
print(people)
```

- A. "Malkovitch" (a string)
- B. Nothing will be output

- C. This code will compile but crash
- D. ["Malkovitch"] (an array)
- E. This code will not compile
- F. ["Malkovitch", "Malkovitch"] (an array)
- G. "Malkovitch", "Malkovitch" (two strings)

## Question 22

What output will be produced by the code below?

```
func greet(_ name: inout String) {  
    name = name.uppercased()  
    print("Greetings, \(name)!")  
}  
  
var name = "Mal"  
greet(name)  
print("Goodbye, \(name)!")
```

- A. This code will compile but crash
- B. "Greetings, MAL!", "Goodbye, Mal!"
- C. "Greetings, MAL!", "Goodbye, MAL!"
- D. Nothing will be output
- E. This code will not compile
- F. "Greetings, Mal!", "Goodbye, MAL!"

## Question 23

What output will be produced by the code below?

```
class Starship {
```

```

var name: String

override init(initialName: String) {
    name = initialName
}

let serenity = Starship(initialName: "Serenity")
print(serenity.name)

```

- A. This code will not compile
- B. nil
- C. ""
- D. Nothing will be output
- E. This code will compile but crash
- F. "Serenity"

## Question 24

When this code is executed, how many items will the **names** array contain?

```

var names = [String]()
names.reserveCapacity(2)
names.append("Amy")
names.append("Rory")
names.append("Clara")

```

- A. This code will compile but crash
- B. This code will not compile
- C. 1
- D. 3
- E. 2

F. 0

## Question 25

What output will be produced by the code below?

```
let numbers = [1, 3, 5, 7, 9]
let result = numbers.map { $0 * 10 }
print(numbers)
```

- A. 10, 30, 50, 70, 90
- B. This code will not compile
- C. Nothing will be output
- D. 1, 3, 5, 7, 9
- E. This code will compile but crash

## Question 26

When this code is executed, how many items will the **result** array contain?

```
let numbers = [1, 3, 5, 7, 9]
let result = numbers.filter { $0 >= 5 }
```

- A. 4
- B. 3
- C. 5
- D. 0
- E. This code will not compile
- F. This code will compile but crash

## Question 27

What output will be produced by the code below?

```
let numbers = Array(1..<10)
print(numbers.count)
```

- A. This code will not compile
- B. Nothing will be output
- C. 8
- D. This code will compile but crash
- E. 9
- F. 10
- G. 0

## Question 28

When this code is executed, what will the **numbers** constant contain?

```
let numbers = [1, 2, 3].map { [$0, $0] }
```

- A. [[1, 1], [1, 2], [1, 3]]
- B. [1, 2, 3]
- C. This code will not compile
- D. This code will compile but crash
- E. [[1, 1], [2, 1], [3, 1]]
- F. [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
- G. [[1, 1], [2, 2], [3, 3]]
- H. [1, 1, 2, 2, 3, 3]

## Question 29

When this code is executed, what will the **numbers** constant contain?

```
let numbers = [1, 2, 3].flatMap { [$0, $0] }
```

- A. [1, 2, 3]
- B. [1, 1, 2, 2, 3, 3]
- C. [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
- D. This code will compile but crash
- E. [[1, 1], [1, 2], [1, 3]]
- F. This code will not compile
- G. [[1, 1], [2, 1], [3, 1]]
- H. [[1, 1], [2, 2], [3, 3]]

## Question 30

What output will be produced by the code below?

```
let userLoggedIn: Bool? = false

if !userLoggedIn! {
    print("Message one")
} else {
    print("Message two")
}
```

- A. "Message two"
- B. "Message one"
- C. Nothing will be output
- D. This code will not compile
- E. This code will compile but crash

## Answers (Intermediate)

**Question 1:** The correct answer is **I**, 'Bones'. The code accesses the "doctor" element in the **names** dictionary, which is not set: dictionaries are case-sensitive. This will cause **nil** to be returned, which then triggers the nil coalescing operator to set **doctor** to be "Bones".

**Question 2:** The correct answer is **D**, 'This code will not compile'. The **sum()** function is declared to take a variadic **Int** parameter, which means it takes zero or more integers. While the body of the function is correct, the way it is called is incorrect because the code should read **sum(numbers: 1, 2, 3, 4, 5)**.

**Question 3:** The correct answer is **E**, 'Criminal masterminds: Malcolm, Kaylee, Zoe'. The **greet()** method is declared as accepting a variadic string parameter, which means it should be called using one or more strings that will be made accessible as an array inside the function. The **greet()** method takes that array, joins it using commas, then prints it out as part of a larger message.

**Question 4:** The correct answer is **A**, 'This code will not compile'. When using **forEach** you are given each item of an array using the value **\$0**. This code incorrectly tries to use **\$1**, and so will not compile.

**Question 5:** The correct answer is **F**, 'This code will not compile'. The **round()** method accepts a **Double** and returns a **Double**, or accepts a **Float** and returns a **Float**. This code creates **rounded** as an integer then tries to force the result of **round()** into it, which is not allowed without a typecast.

**Question 6:** The correct answer is **D**, 'FizzBuzz'. The **fizzbuzz()** method creates a tuple of two booleans. The first boolean is true if the **number** parameter is evenly divisible by 3, and the second is true if it's evenly divisible by 5. The code uses **15**, so the tuple will be **(true, true)**, and thus will print **FizzBuzz**.

**Question 7:** The correct answer is **D**, 'This code will not compile'. The `setName()` method is attempting to change the `name` parameter, which is prohibited unless the `mutating` keyword is used. The correct code should be `mutating func setName(_ newName: String)`.

**Question 8:** The correct answer is **F**, '1, 5, 9, 13'. `stride()` is useful to move through a range of numbers. In this instance, it will count from 1 up to (but not including) 17, incrementing by four each time.

**Question 9:** The correct answer is **B**, 'Neither variable can be assigned to the other'. Even though a `Double` holds more data than a `Float`, Swift's strict type safety means that you can not assign either variable to the other without typecasting.

**Question 10:** The correct answer is **D**, '16'. Swift lets you declare numbers using `0x` as a hexadecimal prefix. `0x10` is 16 in hexadecimal.

**Question 11:** The correct answer is **E**, 'This code will compile but crash'. Adding 1 to the largest integer will cause an integer overflow, which is an exception.

**Question 12:** The correct answer is **D**, 'Int?'. Creating an integer from a string will fail if the string does not contain a valid number. So, this constructor returns `Int?` to give you either a number (on success) or nil (on failure.)

**Question 13:** The correct answer is **A**, '3'. Sets are similar to arrays, although each item can only appear once. If you add an item more than once, it will be silently ignored.

**Question 14:** The correct answer is **C**, '0'. `[]` is shorthand syntax for "an empty dictionary", which causes the `crew` dictionary to be wiped.

**Question 15:** The correct answer is **F**, '1'. Although we've declared the `NSMutableDictionary` to be constant, it's a reference type and so will happily mutate itself regardless of its supposed "constant" status. So, this code will output 1 because the value was added successfully.

**Question 16:** The correct answer is **E**, 'Second group'. Although there are overlapping ranges in this code, Swift will simply find and execute the first one that matches. The first case, `0..<5`, will not be matched because it uses the half-open range operator, which excludes the upper bound.

**Question 17:** The correct answer is **B**, 'X was 556'. Swift will execute the first case block that matches its switch value. In this case, **point** is a tuple with the value **(556, 0)**, so the **(let x, 0)** case will be matched and its message printed.

**Question 18:** The correct answer is **D**, 'This code will not compile'. The **willThrowAnError()** method throws an exception, but it is not marked using the **throws** keyword. The code should read **func willThrowAnError() throws**.

**Question 19:** The correct answer is **A**, 'This code will compile but crash'. The main **repeat** loop is set to loop forever, or at least until someone calls **break** on it. In the code, this happens only when **i % 2** is 0, i.e. when it's an even number, which can never happen because we start with 1 and also add 2. So, even though it will take a long time, the **i** variable will eventually overflow and trigger a crash.

**Question 20:** The correct answer is **B**, 'Crew'. Swift will correctly match the first case again "Simon", then the **fallthrough** keyword will cause control to fall into the next case block and print "Crew".

**Question 21:** The correct answer is **F**, '["Malkovitch", "Malkovitch"] (an array)'. This array constructor automatically creates as many elements as the **count** property requests, with each one being set to the value of **repeating**.

**Question 22:** The correct answer is **E**, 'This code will not compile'. The **greet()** function declares its **name** parameter to be **inout**, which means it must be passed in using **&**. The code should be written **greet(&name)**.

**Question 23:** The correct answer is **A**, 'This code will not compile'. The **init()** method is declared using **override**, but this class does not inherit from anything so the keyword must not be used.

**Question 24:** The correct answer is **D**, '3'. The **reserveCapacity()** method is there to let you give an indication of how many elements you intend to store in an array, but it does not place any sort of limit.

**Question 25:** The correct answer is **D**, '1, 3, 5, 7, 9'. The call to **map** will multiply each integer in the **numbers** array by 10, and assign the result to the **result** constant. However, the value

that is printed is **numbers**, not **result**, so the original integers will be printed.

**Question 26:** The correct answer is **B**, '3'. The call to **filter** will return items from the **numbers** array only if they are greater than or equal to 5.

**Question 27:** The correct answer is **E**, '9'. This array constructor creates values based on the range from 1 up to (but not including 10). This means it will include the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9, so the count will be 9.

**Question 28:** The correct answer is **G**, '[[1, 1], [2, 2], [3, 3]]'. The code loops over every number in the **numbers** array, and creates a new array for each number that contains that number twice. So, 1 will be converted to [1, 1] and so on.

**Question 29:** The correct answer is **B**, '[1, 1, 2, 2, 3, 3]'. A call to **map** code would loop over every number in the **numbers** array, and creates a new array for each number that contains that number twice. So, 1 will be converted to [1, 1] and so on. In this case, we're using **flatMap**, which causes the resulting array of arrays to be flattened to a single array, meaning that [[1, 1]] would become [1, 1] and so on.

**Question 30:** The correct answer is **B**, 'Message one'. The (deliberately clumsy) expression `!userLoggedIn!` means, "force unwrap this boolean, then negate it." The **userLoggedIn** boolean is set to be **false**, so it will be true when negated, meaning that "Message one" will be printed.