HACKING WITH SWIFT

SWIFT CODING CHALLENGES

REAL PROBLEMS, REAL SOLUTIONS

Prepare for iOS interviews,
test yourself against f[illegible]s,
and level up you[illegible].

FREE SAMPLE

Paul Hudson

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

Hint #1: You should work with the **`characters`** property of the input string. This is an array-like object 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.characters {
        if usedLetters.contains(letter) {
            return false
        }

        usedLetters.append(letter)
    }

    return true
}
```

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 **`characters`** property of 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's **`characters`** property if there are no duplicates.

In code you would write this:

```
func challenge1b(input: String) -> Bool {
    return Set(input.characters).count == input.characters.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 arrays using their `reversed()` method.

Hint #2: Two arrays compare as equal if they contain the same items 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: The `characters` property of a string isn't really an array, but you can make it one using `Array()`.

Hint #4: 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 it has a particular quirk in Swift that might have caught you out: the `characters` property of a String might

look a bit like an array, but it's actually a **`String.CharacterView`**. Fortunately, you can create an array from it like this:

```
let characterArray = Array(input.characters)
```

Once you figure that out, you'll just need to use **`reversed()`** to reverse the string's characters, then compare the reversed array with the character view-as-array, like this:

```
func challenge2(input: String) -> Bool {
   return input.characters.reversed() ==
Array(input.characters)
}
```

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 challenge2(input: String) -> Bool {
   let lowercase = input.lowercased()
   return lowercase.characters.reversed() ==
Array(lowercase.characters)
}
```

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 “b 1 a 2” 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.

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: A faster solution is to convert both strings to 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.

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.characters {
        if let index = checkString.characters.index(of: letter) {
            checkString.characters.remove(at: index)
        } else {
            return false
        }
    }

    return checkString.characters.count == 0
}
```

That solution works, but is less than ideal because you're having to look up letter positions repeatedly using **`index(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 more efficient solution is to make arrays out of both sets of string characters, then sort them. Once that's done, you can do a direct comparison using **`==`**. This ends up not only being faster to run, but also involving much less code:

```
func challenge3b(string1: String, string2: String) -> Bool {
    let array1 = Array(string1.characters)
    let array2 = Array(string2.characters)
    return array1.sorted() == array2.sorted()
}
```

For bonus interview points, you might be tempted to write something like this:

```
return array1.count == array2.count && array1.sorted() ==
```

```
array2.sorted()
```

However, there's no need – the **==** operator for arrays does that before doing its item-by-item comparison, so doing it yourself is just redundant.

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 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.

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.characters {
        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.characters.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 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 runs about 10% slower than the first

solution.

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.characters.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.characters.count - modified.characters.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 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 slow 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.characters)
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.characters.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 **Character** array:

```
func challenge6b(string: String) -> String {
    var used = [Character]()

    for letter in string.characters {
```

```
        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.characters.filter {
        used.updateValue(true, forKey: $0) == nil
    }

    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, it's about 3x

slower than the second solution when using our sample input and output data, so although it gets full marks for cleverness it falls short on performance.

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, 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.characters {
        if letter == " " {
            if seenSpace { continue }
            seenSpace = true
        } else {
            seenSpace = false
        }

        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 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 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 double 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.characters.count == rotated.characters.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.