Interview Cake

I figured out how to get rich: online poker.

I *suspect* the online poker game I'm playing shuffles cards by doing a single riffle.⌐

To prove this, **let's write a method to tell us if a full deck of cards `shuffledDeck` is a single riffle of two other halves `half1` and `half2`.**

We'll represent a stack of cards as an array of integers in the range 1..52 (since there are 52 distinct cards in a deck).

Why do I care? A single riffle is *not* a completely random shuffle. If I'm right, I can make more informed bets and get rich and finally prove to my ex that I am not a "loser with an unhealthy cake obsession" (even though it's too late now because she let me go and she's never getting me back).

Gotchas

Watch out for index out of bounds errors! Will your method ever try to grab the 0th item from an empty array, or the nth item from an array with n elements (where the last index would be $n - 1$)?

We can do this in $O(n)$ time and $O(1)$ additional space.

Did you come up with a recursive solution? Keep in mind that you may be incurring a hidden space cost (probably $O(n)$) in the call stack!⌐ You can avoid this using an iterative approach.

Breakdown

How can we re-phrase this problem in terms of smaller subproblems?

Breaking the problem into smaller subproblems will clearly involve reducing the size of at least one of our stacks of cards. So to start, let's try taking the first card out of `shuffledDeck`.

What should be true of this card if `shuffledDeck` is a riffle of `half1` and `half2`?

If `shuffledDeck` is a riffle of `half1` and `half2`, then the first card from `shuffledDeck` should be either the same as the first card from `half1` or the same as the first card from `half2`.

If we're looking at our decks face-up, from above (so we only see the top card), this *could* be a single riffle:
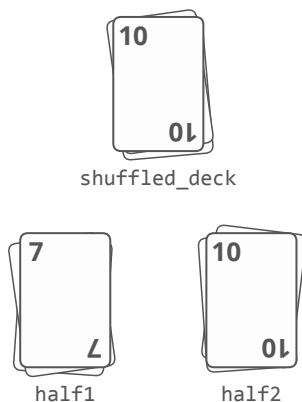
While this could not:
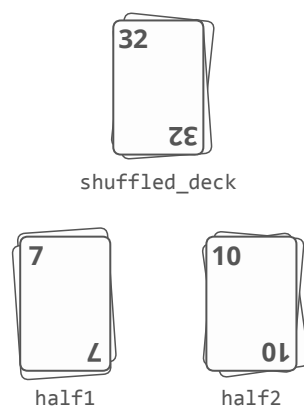
Now that we know the first card checks out, how do we get to our subproblem?

Let's "throw out" the top card from `shuffledDeck` as well as the card it matched with from the top of `half1` or `half2`. Those cards are now "accounted for."

Now we're left with a smaller version of the original problem, which we can solve using the same approach! So we keep doing this over and over until we exhaust `shuffledDeck`. If we get to the end and each card "checks out," we return `true`.

How do we implement this in code?

Now that we have a problem that's the same as the original problem except smaller, our first thought might be to use recursion. All we need is a base case. What's our base case?

We stop when we run out of cards in our shuffledDeck. So that's our base case: when we've checked all cards in shuffledDeck, we return true because we know all of the cards have been "accounted for."

```Java
private static int[] removeTopCard(int[] cards) {
    return Arrays.copyOfRange(cards, 1, cards.length);
}


public static boolean isSingleRiffle(int[] half1, int[] half2, int[] shuffledDeck) {

    // base case
    if (shuffledDeck.length == 0) {
        return true;
    }

    // if the top of shuffledDeck is the same as the top of half1
    // (making sure first that we have a top card in half1)
    if (half1.length > 0 && half1[0] == shuffledDeck[0]) {

        // take the top cards off half1 and shuffledDeck and recurse
        return isSingleRiffle(removeTopCard(half1), half2, removeTopCard(shuffledDeck));

    // if the top of shuffledDeck is the same as the top of half2
    } else if (half2.length > 0 && half2[0] == shuffledDeck[0]) {

        // take the top cards off half2 and shuffledDeck and recurse
        return isSingleRiffle(half1, removeTopCard(half2), removeTopCard(shuffledDeck));

    // top of shuffledDeck doesn't match top of half1 or half2
    // so we know it's not a single riffle
    } else {
        return false;
    }
}
```

This solution will work. But we can do better.

Before we talk about optimization, note that our inputs are of *small* and *constant* size. This method will take hardly any time or space, even if it *could be* more efficient. In industry, especially at small startups that want to move quickly, optimizing this might be considered a "premature optimization." But if we're going to do something inefficient, we should at least *know* about it. Great engineers have both the *skill* to see how to optimize their code and the *wisdom* to know when those optimizations aren't worth it. At this point in the interview I recommend saying "I think we can optimize this a bit further, although given the constraints on the input this probably won't be very resource-intensive anyway...should we talk about optimizations?"

Okay, back to our show. This method will take $O(n^2)$ time and $O(n^2)$ additional space.

Whaaaaat? Yeah. Take a look at this snippet:

```Java
private static int[] removeTopCard(int[] cards) {

    return Arrays.copyOfRange(cards, 1, cards.length);

}


return isSingleRiffle(removeTopCard(half1), half2, removeTopCard(shuffledDeck));
```

In particular this expression:

```Java
Arrays.copyOfRange(cards, 1, cards.length);
```

That's a slice, and it costs $O(m)$ time and space, where m is the size of the resulting array. That's going to determine our overall time and space cost here—the rest of the work we're doing is constant space and time.

In our recursing we'll build up n frames on the call stack. Each of those frames will hold a *different slice* of our original `shuffledDeck` (and `half1` and `half2`, though we only slice one of them in each recursive call).

So, what's the total time and space cost of all our slices?

If `shuffledDeck` has n items to start, taking our first slice takes $n-1$ time and space (plus half that, since we're also slicing one of our halves—but that's just a constant multiplier so we can ignore it). In our second recursive call, slicing takes $n-2$ time and space. Etcetera.

So our total time and space cost for slicing comes to:

$$(n - 1) + (n - 2) + \ldots + 2 + 1$$

This is a <u>common series</u>⌄ that turns out to be $O(n^2)$.

We can do better than this $O(n^2)$ time and space cost. One way we could to that is to avoid slicing and instead keep track of indices in the array:

Java

```java
public static boolean isSingleRiffle(int[] half1, int[] half2, int[] shuffledDeck) {

    return isSingleRiffle(half1, half2, shuffledDeck, 0, 0, 0);

}


private static boolean isSingleRiffle(int[] half1, int[] half2, int[] shuffledDeck,
        int shuffledDeckIndex, int half1Index, int half2Index) {


    // base case we've hit the end of shuffledDeck
    if (shuffledDeckIndex == shuffledDeck.length) {

        return true;

    }


    // if we still have cards in half1
    // and the "top" card in half1 is the same
    // as the top card in shuffledDeck
    if ((half1Index < half1.length)
            && (half1[half1Index] == shuffledDeck[shuffledDeckIndex])) {

        half1Index++;


    // if we still have cards in half2
    // and the "top" card in half2 is the same
    // as the top card in shuffledDeck
    } else if ((half2Index < half2.length)
            && (half2[half2Index] == shuffledDeck[shuffledDeckIndex])) {

        half2Index++;


    // if the top card in shuffledDeck doesn't match the top
    // card in half1 or half2, this isn't a single riffle.
    } else {

        return false;

    }


    // the current card in shuffledDeck has now been "accounted for"
    // so move on to the next one
    shuffledDeckIndex++;
    return isSingleRiffle(half1, half2, shuffledDeck, shuffledDeckIndex, half1Index, half2Index);

}
```

So now we're down to $O(n)$ time, but we're still taking $O(n)$ space in the call stack because of our recursion. We can rewrite this as an iterative method to get that memory cost down to $O(1)$.

What's happening in each iteration of our recursive method? Sometimes we're taking a card off of `half1` and sometimes we're taking a card off of `half2`, but we're *always* taking a card off of `shuffledDeck`.

So what if instead of taking cards off of `shuffledDeck` 1-by-1, we *iterated over them*?

Solution

We walk through `shuffledDeck`, seeing if each card *so far* could have come from a riffle of the other two halves. To check this, we:

1. Keep pointers to the current index in `half1`, `half2`, and `shuffledDeck`.
2. Walk through `shuffledDeck` from beginning to end.
3. If the current `card` in `shuffledDeck` is the same as the top card from `half1`, increment `half1Index` and keep going. This can be thought of as "throwing out" the top cards of `half1` and `shuffledDeck`, reducing the problem to the remaining cards in the stacks.
4. Same as above with half2.
5. If the current `card` isn't the same as the card at the top of `half1` or `half2`, we know we don't have a single riffle.
6. If we make it all the way to the end of `shuffledDeck` and each card checks out, we know we do have a single riffle.

Java ▾

```java
public static boolean isSingleRiffle(int[] half1, int[] half2, int[] shuffledDeck) {

    int half1Index = 0;
    int half2Index = 0;

    int half1MaxIndex = half1.length - 1;
    int half2MaxIndex = half2.length - 1;


    for (int card : shuffledDeck) {


        // if we still have cards in half1
        // and the "top" card in half1 is the same
        // as the top card in shuffledDeck
        if (half1Index <= half1MaxIndex
                && card == half1[half1Index]) {
            half1Index++;


        // if we still have cards in half2
        // and the "top" card in half2 is the same
        // as the top card in shuffledDeck
        } else if (half2Index <= half2MaxIndex
                && card == half2[half2Index]) {
            half2Index++;


        // if the top card in shuffledDeck doesn't match the top
        // card in half1 or half2, this isn't a single riffle.
        } else {
            return false;
        }
    }


    // all cards in shuffledDeck have been "accounted for"
    // so this is a single riffle!
    return true;
}
```

Complexity

$O(n)$ time and $O(1)$ additional space.

Becky if you're reading this I didn't really mean what I said in the problem statement. It's just that things have been hard lately and anyway if you'll just give me another chance I promise it won't be like last time. I'm a wreck without you. Like a collapsed soufflé. Please Becky.

Bonus

1. This assumes `shuffledDeck` contains all 52 cards. What if we can't trust this (e.g. some cards are being secretly *removed* by the shuffle)?
2. This assumes each number in `shuffledDeck` is unique. How can we adapt this to rifling arrays of random integers with *potential repeats*?
3. Our solution returns `true` if you just cut the deck—take one half and put it on top of the other. While that *technically* meets the definition of a riffle, what if you wanted to *ensure* that some mixing of the two halves occurred?

What We Learned

If you read the whole breakdown section, you might have noticed that we started with a recursive method that looked simple but turned out to cost $O(n^2)$ time and space because each recursive step *created its own new "slice"* of the input array. If you missed that part, go back and take a look.

Be careful of the hidden time and space costs of array slicing! Consider tracking array indices "by hand" instead (as we do in our final solution).

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.