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 function to tell us if a full deck of cards `shuffled_deck` 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 function 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 `shuffled_deck`.

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

If `shuffled_deck` is a riffle of `half1` and `half2`, then the first card from `shuffled_deck` 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:
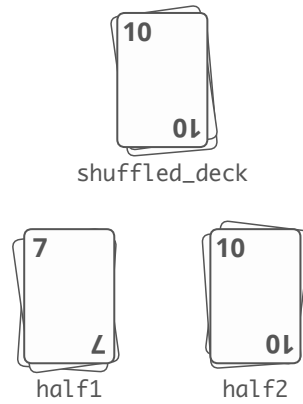
While this could not:
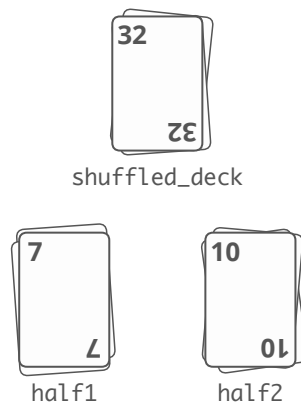
Now that we know the first card checks out, how do we get to our subproblem?

Let's "throw out" the top card from `shuffled_deck` 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 `shuffled_deck`. 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 `shuffled_deck`. So that's our base case: when we've checked all cards in `shuffled_deck`, we return `true` because we know all of the cards have been "accounted for."

```ruby
def is_single_riffle_recursive(half1, half2, shuffled_deck)

    # base case
    if shuffled_deck.length == 0
        return true
    end

    # if the top of shuffled_deck is the same as the top of half1
    # (making sure first that we have a top card in half1)
    if half1.length > 0 and half1[0] == shuffled_deck[0]

        # take the top cards off half1 and shuffled_deck and recurse
        return is_single_riffle_recursive(half1[1..-1], half2, shuffled_deck[1..-1])

    # if the top of shuffled_deck is the same as the top of half2
    elsif half2.length > 0 and half2[0] == shuffled_deck[0]

        # take the top cards off half2 and shuffled_deck and recurse
        return is_single_riffle_recursive(half1, half2[1..-1], shuffled_deck[1..-1])

    # top of shuffled_deck doesn't match top of half1 or half2
    # so we know it's not a single riffle
    else
        return false
    end
end
```

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 function 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 function will take $O(n^2)$ time and $O(n^2)$ additional space.

Whaaaaat? Yeah. Take a look at this snippet:

```Ruby
return is_single_riffle_recursive(half1[1..-1], half2, shuffled_deck[1..-1])
```

In particular this expression:

```Ruby
half1[1..-1]
```

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 shuffled_deck (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 shuffled_deck 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:

$$(n - 1) + (n - 2) + ... + 2 + 1$$

This is a common series⌐ 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:

```
def is_single_riffle_recursive_optimized(half1, half2, shuffled_deck, shuffled_deck_index=0, hal

    # base case we've hit the end of shuffled_deck
    if shuffled_deck_index == shuffled_deck.length
        return true
    end

    # if we still have cards in half1
    # and the "top" card in half1 is the same
    # as the top card in shuffled_deck
    if (half1_index < half1.length) and \
            half1[half1_index] == shuffled_deck[shuffled_deck_index]
        half1_index += 1

    # if we still have cards in half2
    # and the "top" card in half2 is the same
    # as the top card in shuffled_deck
    elsif (half2_index < half2.length) and \
            half2[half2_index] == shuffled_deck[shuffled_deck_index]
        half2_index += 1

    # if the top card in shuffled_deck doesn't match the top
    # card in half1 or half2, this isn't a single riffle.
    else
        return false
    end

    # the current card in shuffled_deck has now been "accounted for"
    # so move on to the next one
    shuffled_deck_index += 1
    return is_single_riffle_recursive_optimized(half1, half2, shuffled_deck, shuffled_deck_index
end
```

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 function to get that memory cost down to $O(1)$.

What's happening in each iteration of our recursive function? 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 shuffled_deck.

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

Solution

We walk through `shuffled_deck`, 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 `shuffled_deck`.
2. Walk through `shuffled_deck` from beginning to end.
3. If the current `card` in `shuffled_deck` is the same as the top card from `half1`, increment `half1_index` and keep going. This can be thought of as "throwing out" the top cards of `half1` and `shuffled_deck`, 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 `shuffled_deck` and each card checks out, we know we do have a single riffle.

```ruby
def is_single_riffle(half1, half2, shuffled_deck)
    half1_index = 0
    half2_index = 0
    half1_max_index = half1.length - 1
    half2_max_index = half2.length - 1

    shuffled_deck.each do |card|
        # if we still have cards in half1
        # and the "top" card in half1 is the same
        # as the top card in shuffled_deck
        if half1_index <= half1_max_index and \
                card == half1[half1_index]
            half1_index += 1

        # if we still have cards in half2
        # and the "top" card in half2 is the same
        # as the top card in shuffled_deck
        elsif half2_index <= half2_max_index and \
                card == half2[half2_index]
            half2_index += 1

        # if the top card in shuffled_deck doesn't match the top
        # card in half1 or half2, this isn't a single riffle.
        else
            return false
        end
    end

    # all cards in shuffled_deck have been "accounted for"
    # so this is a single riffle!
    return true
end
```

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 `shuffled_deck` 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 `shuffled_deck` 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 function 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).