You're working on a secret team solving coded transmissions.

Your team is scrambling to decipher a recent message, worried it's a plot to break into a major European National Cake Vault. The message has been *mostly* deciphered, but all the words are backwards! Your colleagues have handed off the last step to you.

Write a function `reverseWords()` that takes a string `message` and reverses the order of the words in-place .

> Since strings in Java are **immutable** , we'll first convert the string into an array of characters, do the in-place word reversal on that array, and re-join that array into a string before returning it. This isn't technically "in-place" and the array of characters will cost $O(n)$ additional space, but it's a reasonable way to stay within the spirit of the challenge. If you're comfortable coding in a language with mutable strings, that'd be even better!

For example:

```java
String message = "find you will pain only go you recordings security the into if";


reverseWords(message);
// returns: "if into the security recordings you go only pain will you find"
```

When writing your function, assume the message contains only letters and spaces, and all words are separated by one space.

Gotchas

Are you sure you're operating on the array of characters *in-place*?

We can do this in $O(n)$ time.

If you're swapping individual words one at a time, consider what happens when the words are different lengths. Isn't *each swap* $O(n)$ time in the worst case?

Breakdown

We'll start by converting the message string into an array of characters.

```java
public String reverseWords(String message) {
    char[] messageChars = message.toCharArray();
    ...
}
```

Now, how can we move things around?

Let's start with a simpler problem. What if we wanted to **reverse all the characters** in the message?

Well, we could swap the first character with the last character, then the second character with the second to last character, and so on, moving towards the middle. Can you implement this in code?

```java
public String reverseCharacters(String message) {

    char[] messageChars = message.toCharArray();

    // walk towards the middle, from both sides
    for (int frontIndex = 0; frontIndex < messageChars.length / 2; frontIndex++) {
        int backIndex = messageChars.length - frontIndex - 1;

        // swap the front char and back char
        char tempChar = messageChars[frontIndex];
        messageChars[frontIndex] = messageChars[backIndex];
        messageChars[backIndex]  = tempChar;
    }

    return new String(messageChars);
}
```

Ok, looks good. **Does this help us?**

Can we use the same concept but apply it to *words* instead of *characters*?

Probably. We'll have to figure out a couple things:

1. How do we figure out where words begin and end?
2. Once we know the start and end indices of two words, how do we *swap* those two words?

We could attack either of those first, but I'm already seeing a potential problem in terms of runtime. Can you guess what it is?

What happens when you swap two words that *aren't* the same length?

```Java
"the eagle has landed"
```

Supposing we already knew the start and end indices of 'the' and 'landed', how long would it take to swap them?

$O(n)$ time, where n is the total length of the string!

Why? Notice that in addition to moving 'the' to the back and moving 'landed' to the front, we have to "scoot over" *everything in between*, since 'landed' is longer than 'eagle'.

So what'll be the *total* time cost with this approach? Assume we'll be able to learn the start and end indices of all of our words in just one pass ($O(n)$ time).

$O(n^2)$ total time. Why? In the worst case we have almost as many words as we have characters, and we're always swapping words of different lengths. For example:

```Java
"a bb c dd e ff g hh"
```

We take $O(n)$ time to swap the first and last words (we have to move all n characters):

```Java
"a bb c dd e ff g hh" // input
"hh bb c dd e ff g a" // first swap
```

Then for the second swap:

```
"a bb c dd e ff g hh" // input
"hh bb c dd e ff g a" // first swap
"hh g c dd e ff bb a" // second swap
```

We have to move all n characters *except* the first and last words, and a couple spaces. So we move $n - 5$ characters in total.

For the third swap, we have another 5 characters we don't have to move. So we move $n - 10$ in total. We'll end up with a series like this:

$$n + (n - 5) + (n - 10) + (n - 15) + ... + 6 + 1$$

This is a subsection of the common triangular series . We're just skipping 4 terms between each term!

So we have the sum of "every fifth number" from that triangular series. That means our sum will be about a fifth the sum of the original series! But in big O notation dividing by 5 is a constant, so we can throw it out. The original triangular series is $O(n^2)$, and so is our series with every fifth element!

Okay, so $O(n^2)$ time. That's pretty bad. It's *possible* that's the best we can do...but maybe we can do better?

Let's try manipulating a sample input by hand.

And remember what we did for our character-level reversal...

Look what happens when we do a character-level reversal:

```Java
"the eagle has landed" // input
"dednal sah elgae eht" // character-reversed
```

Notice anything?

What if we put it up against the desired output:

```Java
"the eagle has landed" // input
"dednal sah elgae eht" // character-reversed
"landed has eagle the" // word-reversed (desired output)
```

The character reversal reverses the words! It's a great first step. From there, we just have to "unscramble" each word.

More precisely, we just have to re-reverse each word!

Solution

We'll write a helper function `reverseCharacters()` that reverses all the characters in a string between a `frontIndex` and `backIndex`. We use it to:

1. Reverse **all the characters in the entire message**, giving us the correct *word order* but with *each word backwards*.
2. Reverse **the characters in each individual word**.

Java

```
public String reverseWords(String message) {

    char[] messageChars = message.toCharArray();

    // first we reverse all the characters in the entire messageChars array
    reverseCharacters(messageChars, 0, messageChars.length - 1);
    // this gives us the right word order
    // but with each word backwards

    // now we'll make the words forward again
    // by reversing each word's characters

    // we hold the index of the *start* of the current word
    // as we look for the *end* of the current word
    int currentWordStartIndex = 0;
    for (int i = 0; i <= messageChars.length; i++) {

        // found the end of the current word!
        if (i == messageChars.length || messageChars[i] == ' ') {

            // if we haven't exhausted the string our
            // next word's start is one character ahead
            reverseCharacters(messageChars, currentWordStartIndex, i - 1);
            currentWordStartIndex = i + 1;
        }
    }

    return new String(messageChars);
}
```

```
public void reverseCharacters(char[] messageChars, int startIndex, int endIndex) {

    // walk towards the middle, from both sides
    while (startIndex < endIndex) {

        // swap the front char and back char
        char temp = messageChars[startIndex];
        messageChars[startIndex] = messageChars[endIndex];
        messageChars[endIndex] = temp;
        startIndex++;
        endIndex--;
    }
}
```

Complexity

$O(n)$ time and $O(n)$ space. Our space cost comes from converting the message string to an array. If our input was an array, our space cost would be $O(1)$ because we'd be using a constant amount of additional space beyond the input.

Hmm, the team used your function to finish deciphering the message. There definitely seems to be a plot brewing, but no specifics on where. The sender seemed to really like Eccles cakes. Any ideas?

Bonus

How would you handle punctuation?

What We Learned

The naive solution of reversing the words one at a time had a worst-case $O(n^2)$ runtime. That's because swapping words with *different lengths* required "scooting over" all the other characters to make room.

To get around this "scooting over" issue, we reversed all the *characters* in the string first. This put all the words in the correct spots, but with the characters in each word backwards. So to get the final answer, we reversed the characters in each word. This all takes two passes through the string, so $O(n)$ time total.

This might seem like a blind insight, but we derived it by using a clear strategy:

Solve a *simpler* version of the problem (in this case, reversing the characters instead of the words), and see if that gets us closer to a solution for the original problem.

We talk about this strategy in the "get unstuck" section of our coding interview tips (/article/coding-interview-tips#unstuck).