Palindrome Lab Solution 1

(https://github.com/learn-co-curriculum/phase-1-algorithms-palindrome-solution-1) (https://github.com/learn-co-curriculum/phase-1-algorithms-palindrome-solution-1/issues/new)

Learning Goals

- Practice algorithmic problem solving
- Evaluate multiple solutions to a problem

Introduction

Now that you've had a chance to work through the palindrome lab, we'll walk through a couple different solutions to this problem. As you'll see, depending on your initial approach and what kind of pseudocode you came up with, you may end up with entirely different algorithms. You may have even come up with something different from the solutions we'll review here, and that's ok! We'll talk about how to evaluate different solutions and discuss their pros and cons at the end.

For each solution, we'll work through the steps and discuss our problem solving process as well so you can see how we ended up with these particular algorithms.

As a reminder, here are the instructions:

> Write a function `isPalindrome` that will receive one argument, a string. Your function should return `true` if the string is a palindrome (that is, if it reads the same forwards and backwards, like `"mom"` or `"racecar"`), and return `false` if it is not a palindrome.
>
> To keep things simple, your function only needs to deal with lowercase strings that are all letters (don't worry about spaces or special characters).

Solution 1

Video Walkthrough

1. Rewrite the Problem in Your Own Words

We'll start by rewriting the problem in a different way to make sure we understand it:

> I need to make an `isPalindrome` function that returns either `true` or `false`. When the input string is the same forwards and backwards, I should return `true`. That means that if the input string is the same after I **reverse** it, I should return true. For instance, `"mom"` in reverse is also `"mom"`, and `"racecar"` in reverse is also `"racecar"`, so my solution should return `true` for these cases. `"hi"` in reverse is `"ih"`, so my solution should return `false` for this case.

Note that this description of the problem highlights the inputs and output (return value), and gives us some ideas to explore later in our pseudocode.

2. Write Your Own Test Cases

Next, let's write a few test cases. The instructions gave us a few examples, but it's also important to come up with our own so we're sure our algorithm handles many different cases. Consider what **edge cases (https://www.geeksforgeeks.org/dont-forget-edge-cases/)** might come up. Can our algorithm handle palindromes that are an even number of letters and an odd number of letters? What about one-letter words?

We can add this code to the `index.js` file to test our function later:

```
if (require.main === module) {
  console.log("Expecting: true");
  console.log("=>", isPalindrome("racecar"));

  console.log("");

  console.log("Expecting: true");
  console.log("=>", isPalindrome("mom"));

  console.log("");

  console.log("Expecting: true");
  console.log("=>", isPalindrome("abba"));

  console.log("");

  console.log("Expecting: true");
  console.log("=>", isPalindrome("a"));

  console.log("");

  console.log("Expecting: false");
  console.log("=>", isPalindrome("hi"));

  console.log("");

  console.log("Expecting: false");
  console.log("=>", isPalindrome("robot"));
}
```

3. Pseudocode

Now that we understand the problem and have a sense of some cases our algorithm can handle, we can write some pseudocode as an intermediate step before writing out our solution. We can refer back to the language we used when rewriting the problem in our own words to help come up with the pseudocode:

> If the input string is the same after I reverse it, I should return true

Here's how we can translate that into a pseudocode version of our algorithm:

```
reverse the input string

if the reversed string is the same as the input
  return true
else
  return false
```

With that in mind, let's write out our solution!

4. Code

Here's the code we are given to start, with the pseudocode added in as comments:

```
function isPalindrome(word) {
  // reverse the input string
  // if the reversed string is the same as the input
  //   return true
  // else
  //   return false
}
```

The one tricky part will be reversing the input string; once we have that, the rest of the function will fall into place easily. Let's start by filling in what we know, and create a **stub** (https://en.wikipedia.org/wiki/Method_stub) for a `reverseString` helper function to come back to later:

```
function reverseString(word) {
  // TODO: implement string reversing functionality
  return word;
}

function isPalindrome(word) {
  // reverse the input string
  const reversedWord = reverseString(word);
  // if the reversed string is the same as the input
  if (word === reversedWord) {
    return true;
  } else {
    return false;
  }
}
```

Writing your code this way in an interview setting is a great way to break up the problem to smaller pieces so you can focus on each step of the logic on its own.

Let's tackle the last part of this problem: reversing the string in our `reverseString` function. If you're not sure how to do this, now's an appropriate time to use Google to help, if the interviewer allows it. Some languages, like Ruby, have built-in methods for reversing strings, but JavaScript doesn't. However, JavaScript *does* have a method for reversing **arrays**, so for our `reverseString` implementation, we'll need to do the following:

```
create an array from the input string
reverse the array
create a string from the reversed array
return the reversed string
```

Here's how we can implement that using some built-in JavaScript methods:

```
function reverseString(word) {
  // create an array from the input string
  const wordArray = word.split("");
  // reverse the array
  const reversedWordArray = wordArray.reverse();
  // create a string from the reversed array
  const reversedWord = reversedWordArray.join("");
  // return the reversed string
  return reversedWord;
}
```

We can also test our `reverseString` function to ensure it does what we expect, so that we can use it with confidence in our `isPalindrome` function:

```
console.log("Expecting: ih");
console.log("=>", reverseString("hi"));

console.log("");

console.log("Expecting: tobor");
console.log("=>", reverseString("robot"));

console.log("");

console.log("Expecting: mom");
console.log("=>", reverseString("mom"));
```

Here's what our completed code looks like all together:

```
function reverseString(word) {
  // create an array from the input string
  const wordArray = word.split("");
  // reverse the array
  const reversedWordArray = wordArray.reverse();
  // create a string from the reversed array
  const reversedWord = reversedWordArray.join("");
  // return the reversed string
  return reversedWord;
}

function isPalindrome(word) {
  // reverse the input string
  const reversedWord = reverseString(word);
  // if the reversed string is the same as the input
  if (word === reversedWord) {
    return true;
  } else {
    return false;
  }
}
```

Now's a good time to check if our implementation passes all of our test cases. Running `node index.js` will check that all the tests cases match our expectations. Now it's time to refactor!

5. Make It Clean and Readable

Reviewing our completed solution should reveal where we can tidy up our code. This step is a bit subjective. The goal isn't to make our code as short as possible — having more lines of code that express your intent is just fine! But we should look for *redundant* code and strive for readability, so that our solution is clear to us and any other developers looking at it for the first time.

One bit of code we can clean up is the `if/else` statement. Since `===` will evaluate to either `true` or `false`, we can simply return the result of that comparison:

```
function isPalindrome(word) {
  // reverse the input string
  const reversedWord = reverseString(word);
  // compare the reversed string to the input
  return word === reversedWord;
}
```

We can also save on a few variable declarations by using method chaining in our `reverseString` function:

```
function reverseString(word) {
  return word.split("").reverse().join("");
}
```

Again, whether or not you choose to refactor this way is a somewhat subjective decision — if you find the prior version of this function easier to understand, then there's no need to go through the additional steps to refactor!

6. Optimize

For this final step, let's talk through a couple considerations for our code's performance. We'll talk about these concepts in more detail in future lessons, but to start with, let's think about our program's:

- Time complexity (how long our algorithm will take to run)
- Space complexity (how much memory our algorithm will use)

When considering performance, it's helpful to think about the worst case scenario. What will happen if our input is five thousand (or five million!) characters instead of just five? Which parts of our code are affected by the size of the input?

Well, in our algorithm, the `reverseString` function is doing most of the heavy lifting. It needs to:

- Split the string

    - That takes up more memory to store the new array, as well as the time it takes JavaScript to iterate over each character of the string to produce a new array
- Reverse the array
    - That takes up more memory to store the reversed array, as well as the time it takes JavaScript to iterate over the array to produce a reversed array
- Join the string
    - That takes up more memory to store the new string, as well as the time it takes JavaScript to iterate over the array to produce a string

Optimizing our solution would mean either coming up with a new way to reverse a string that involves fewer steps, or coming up with another solution that doesn't involve reversing a string. That being said, the solution we came up with handles all the test cases and is well-factored, so even though we could optimize its performance, it's a very fine solution!

Conclusion

In this lesson, we applied a problem solving process to one specific algorithm problem. In the next lesson, we'll work through an alternate solution and compare the two solutions we came up with from a performance and readability standpoint.