Palindrome Lab Solution 2

(https://github.com/learn-co-curriculum/phase-1-algorithms-palindrome-solution-2) (https://github.com/learn-co-curriculum/phase-1-algorithms-palindrome-solution-2/issues/new)

Learning Goals

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

Introduction

Let's work through another solution to the palindrome problem. As you'll see, our initial approach (the way we understand the problem and write the pseudocode for it) will go a long way towards determining the outcome — that's why it's so crucial to spend time on these steps before writing any code!

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 2

Video Walkthrough

1. Rewrite the Problem in Your Own Words

Just like before, 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 if the **first letter** is the same as the **last letter**, and the **second letter** is the same as the **second to last** letter, and so on (until the middle of the word), the function should return `true`.
>
> For the word `"racecar"`, the first and last letter is "r", the second and second to last is "a", the third and third to last is "c", and the middle is "e", so I should return `true`. For the word "robot", the first letter is "r" and the last letter is "t", so I should return `false`.

Note that this description of the problem still highlights the inputs and outputs, but compared to our last description of "reversing the word", the way we understand the problem is significantly different, and will impact our approach when it's time to write the code.

2. Write Your Own Test Cases

We can reuse the same test cases we came up with for the previous solution:

```
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

Our next step, once again, is to write some pseudocode based on our understanding of the problem. The key part of our description is:

> If the **first letter** is the same as the **last letter**, and the **second letter** is the same as the **second to last** letter, and so on (until the middle of the word), the function should return `true`.

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

```
iterate from the beginning of the string to the middle of the string
  compare the letter we're iterating over to the corresponding letter at the end of the string
    if the letters don't match, return false

if we reach the middle, and all the letters match, return true
```

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) {
  // iterate from the beginning of the string to the middle of the string
  // compare the letter we're iterating over to the corresponding letter at the end of the string
  // if the letters don't match, return false
  // if we reach the middle, and all the letters match, return true
}
```

Let's take this one step at a time. First, we need a way to iterate from the beginning of the string to the middle. We can identify the middle of the string using its length:

```
function isPalindrome(word) {
  // iterate from the beginning of the string to the middle of the string
  for (let i = 0; i < word.length / 2; i++) {
    // compare the letter we're iterating over to the corresponding letter at the end of the string
```

```
      // if the letters don't match, return false
    }

    // if we reach the middle, and all the letters match, return true
  }
```

Since this is a slightly uncommon way to use a `for` loop, let's visualize what this loop will do given a few different inputs:

```
Input: "racecar"
Input length divided by two: 3.5
Iteration:
  Index 0 (less than 3.5, keep iterating)
  Index 1 (less than 3.5, keep iterating)
  Index 2 (less than 3.5, keep iterating)
  Index 3 (less than 3.5, keep iterating)
  Index 4 (not less than 3.5, stop iterating)
For "racecar", our loop will iterate up to the middle "e"

Input: "abba"
Input length divided by two: 2
Iteration:
  Index 0 (less than 2, keep iterating)
  Index 1 (less than 2, keep iterating)
  Index 2 (not less than 2, stop iterating)
For "abba", our loop will iterate up to the first "b"
```

Great! This loop will get us up to the middle of the word, so we'll be able to access each letter from the beginning of the word. Next, let's find a way to access the corresponding character from the end of the word. Our goal is to compare the two letters, so we'll want to do something like this, where `i` is our index from the beginning of the word and `j` is the corresponding index from the end of the word:

```
r a c e c a r
0 1 2 3 4 5 6
```

```
i           j

r a c e c a r
0 1 2 3 4 5 6
  i       j

r a c e c a r
0 1 2 3 4 5 6
    i   j

r a c e c a r
0 1 2 3 4 5 6
      ij
```

To calculate `j`, we can use the length of the word minus one to get the last letter of the word, and then subtract `i`, so as `i` increases, `j` will decrease:

```
function isPalindrome(word) {
  // iterate from the beginning of the string to the middle of the string
  for (let i = 0; i < word.length / 2; i++) {
    // compare the letter we're iterating over to the corresponding letter at the end of the string
    const j = word.length - 1 - i;

    // if the letters don't match, return false
  }

  // if we reach the middle, and all the letters match, return true
}
```

Now that we can access the corresponding letters from the beginning and end of the string, we just need to check if they match:

```
function isPalindrome(word) {
  // iterate from the beginning of the string to the middle of the string
```

```
  for (let i = 0; i < word.length / 2; i++) {
    // compare the letter we're iterating over to the corresponding letter at the end of the string
    const j = word.length - 1 - i;
    if (word[i] !== word[j]) {
      // if the letters don't match, return false
      return false;
    }
  }

  // if we reach the middle, and all the letters match, return true
}
```

Now if any letters **don't** match, we'll stop looping and exit the function early with a return value of `false`. Our last step is to return `true` once we reach the end of the loop, since at that point we'll have compared all the letters:

```
function isPalindrome(word) {
  // iterate from the beginning of the string to the middle of the string
  for (let i = 0; i < word.length / 2; i++) {
    // compare the letter we're iterating over to the corresponding letter at the end of the string
    const j = word.length - 1 - i;
    if (word[i] !== word[j]) {
      // if the letters don't match, return false
      return false;
    }
  }

  // if we reach the middle, and all the letters match, return true
  return true;
}
```

It'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.

5. Make It Clean and Readable

Let's do a quick review of our code: are there any variables that could be named more clearly? Is there any redundant or unnecessary code? Can we convert any code to a separate function?

Since our implementation here is already pretty short and sweet, let's just come up with some more descriptive variable names to better express our code's intent. `i` and `j` could be renamed as `startIndex` and `endIndex`, for example:

```
function isPalindrome(word) {
  // iterate from the beginning of the string to the middle of the string
  for (let startIndex = 0; startIndex < word.length / 2; startIndex++) {
    // compare the letter we're iterating over to the corresponding letter at the end of the string
    const endIndex = word.length - 1 - startIndex;
    if (word[startIndex] !== word[endIndex]) {
      // if the letters don't match, return false
      return false;
    }
  }

  // if we reach the middle, and all the letters match, return true
  return true;
}
```

Other than that change, our solution looks pretty good!

6. Optimize

For this final step, let's talk through a couple considerations for our code's performance. Once again, 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)

Again, think about the worst case: how well would our algorithm handle very large input strings?

In the example above, we create two new variables, `startIndex` and `endIndex`, both of which hold numbers, so we aren't adding too much space complexity (numbers are simpler to store in memory than strings and arrays).

Since we are only iterating over half of the length of the string at most, our program's time complexity grows in proportion to the length of the input (for a 10 letter string, we'll need 5 iterations; for a 1000 letter string, we'll need 500 iterations).

This seems like a pretty optimal solution! How could we improve it? Well, what if we had a list of every single possible palindrome in existence, and could immediately check if our input string was in that list? That would certainly improve the time complexity of our algorithm, but we'd need a whole lot of space to store all the possible palindromes. Let's see how it compares to our previous solution as well.

Solution 1 vs Solution 2

Let's take a look at these two solutions and talk about their pros and cons.

Solution 1:

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

function isPalindrome(word) {
  const reversedWord = reverseString(word);
  return word === reversedWord;
}
```

- **Readability**: the code is clear and easy to follow, and expresses its intent using well-named variables and helper functions
- **Performance**: as discussed previously, it's not the most efficient in terms of time and space complexity, since it involves creating additional arrays and strings

Solution 2:

```
function isPalindrome(word) {
  for (let startIndex = 0; startIndex < word.length / 2; startIndex++) {
```

```
    const endIndex = word.length - 1 - startIndex;
    if (word[startIndex] !== word[endIndex]) {
      return false;
    }
  }

  return true;
}
```

- **Readability**: it's more challenging at a glance to tell what this function does, particularly without comments
- **Performance**: this code is more efficient with regards to time and space complexity, since it creates fewer new variables and only iterates through half of the string

All in all, both solutions would be deserving of a positive review from a potential interviewer; and if you're able to discuss the pros and cons of either of these solutions, you're in great shape!

Conclusion

We've spent a long time reviewing solutions to just one relatively simple problem. By spending this extra time and being deliberate about the problem solving process, you'll be better equipped to identify possible solutions, implement them, and also discuss their pros and cons. In the next lessons, we'll discuss a more formal process for determining the time and space complexity of algorithms and introduce the concept of Big O notation.

Resources

Here are a few solutions to this problem from external sources to see alternate approaches.

> **Note**: Some of these examples involve handling more edge cases, like spaces and uppercase characters, which may come up in technical interviews as well.

- **whatsdev (https://www.youtube.com/watch?v=hvV48xfwZCs)**

(https://www.youtube.com/watch?v=hvV48xfwZCs)

- Steve Griffith (https://www.youtube.com/watch?v=saj9KQ3wGtc)

(https://www.youtube.com/watch?v=saj9KQ3wGtc)

- James Q. Quick (https://www.youtube.com/watch?v=5eSJH2CbDBw)

(https://www.youtube.com/watch?v=5eSJH2CbDBw)

- Useful Programmer (https://www.youtube.com/watch?v=jCfvMmIMhFU)

(https://www.youtube.com/watch?v=jCfvMmIMhFU)