Write an efficient function that checks whether any permutation⤵

A permutation is an *ordering* of a set of items.

Example: all permutations of 'tom':

- tom
- tmo
- omt
- otm
- mto
- mot

Not to be confused with a *combination*, which is an *unordered* set or subset.

# of an input string is a palindrome.⤵

A palindrome is a string that's the same when read forward and backward.

Examples:

- civic
- mom
- anna

You can assume the input string only contains lowercase letters.

Examples:

- "civic" should return `true`
- "ivicc" should return `true`
- "civil" should return `false`
- "livci" should return `false`

> **"But 'ivicc' isn't a palindrome!"**
>
> If you had this thought, read the question again carefully. We're asking if any *permutation* of the string is a palindrome. Spend some extra time ensuring you fully understand the question before starting. Jumping in with a flawed understanding of the problem doesn't look good in an interview.

Gotchas

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

Breakdown

The brute force⤵

> A **brute force** algorithm simply tries *all* possible answers to a question and checks them for correctness.
>
> It's rarely the most efficient approach, but it can be helpful to consider the time cost of the brute force approach when building an optimized solution. If your solution isn't faster than the brute force approach, it may not be optimal.

approach would be to **check every permutation of the input string to see if it is a palindrome**.

What would be the time cost? For a string of length n, there are $n!$ permutations (n choices for the first character, times $n-1$ choices for the second character, etc). Checking each length-n permutation to see if it's a palindrome involves checking each character, taking $O(n)$ time. **That gives us $O(n!n)$ time overall. We can do better!**

Let's try **rephrasing the problem**. How can we tell if any permutation of a string is a palindrome?

Well, how would we *check that a string is a palindrome*? We could **use the somewhat-common "keep two pointers" pattern**. We'd start a pointer at the beginning of the string and a pointer at the end of the string, and check that the characters at those pointers are equal as we walk both pointers towards the middle of the string.

```
civic
^   ^

civic
 ^ ^

civic
  ^
```

Can we adapt the idea behind this approach to checking if *any permutation* of a string is a palindrome?

Notice: we're essentially checking that **each character left of the middle has a corresponding copy right of the middle**.

We can simply check that each character appears an even number of times (unless there is a middle character, which can appear once or some other odd number of times). This ensures that the characters can be ordered so that each char on the left side of the string has a matching char on the right side of the string.

But we'll need a data structure to keep track of the number of times each character appears. What should we use?

We could use a hash. (**Tip:** using a hash is *the* most common way to get from a brute force approach to something more clever. It should always be your first thought.)

So we'll go through all the characters and track how many times each character appears in the input string. Then we just have to make sure *no more than one of the characters* appears an odd numbers of times.

That will give us a runtime of $O(n)$, which is the best we can do since we have to look at a number of characters dependent on the length of the input string.

Ok, so we've reached our best run time. But can we still clean our solution up a little?

We don't really care *how many times* a character appears in the string, we just need to know *whether the character appears an* ***even or odd*** *number of times.*

What if we just track whether or not each character appears an odd number of times? Then we can map characters to *booleans*. This will be more explicit (we don't have to check each number's parity, we already have booleans) and we'll avoid the risk of integer overflow.

> When you create an integer variable, your computer allocates a fixed number of bits for storing it. Most modern computers use 32 or 64 bits. But some numbers are *so big* they don't fit even in 64 bits, like sextillion (a *billion trillion*), which is 70 digits in binary.
>
> Sometimes we might have a number that *does* fit in 32 or 64 bits, but if we add to it (or multiply it by something, or do another operation) the result might *not fit* in the original 32 or 64 bits. This is called an **integer overflow**.
>
> For example, let's say we have just **2** bits to store integers with. So we can only hold the unsigned (non-negative) integers 0-3 in binary:
>
> ```
> 00 (0)
> 01 (1)
> 10 (2)
> 11 (3)
> ```
>
> What happens if we have 3 (11) and we try to add 1 (01)? The answer is 4 (100) but that requires 3 bits and we only have 2.
>
> What happens next depends on the language:
>
> - Some languages, like Python or Ruby, will notice that the result won't fit and automatically allocate space for a larger number.
> - In other languages, like C or Java, the processor will sort of "do its best" with the bits it has, taking the true result and throwing out any bits that don't fit. So in our example above, when adding 01 to 11, the processor would take the true result 100 and throw out the highest bit, leaving 00.
> - Swift will throw an error if an integer overflows, unless you've explicitly indicated that overflowing integers should be truncated to fit (like in C or Java).
>
> In languages where integer overflow can occur, you can reduce its likelihood by using larger integer types, like like C's `long long int` or Java's `long`. If you need to store something even bigger, there are libraries built to handle arbitrarily large numbers.

> In some languages, you can also take advantage of overflow-checking features provided by the compiler or interpreter.

if some characters appear a high number of times.

Can we take this a step *further* and clean it up even more?

Even more specifically than *whether characters appear an even or odd number of times*, we really just need to know *there isn't more than one character that appears an odd number of times*.

What if we only track the characters that appear an odd number of times? Is there a data structure even simpler than a hash we could use?

We could use a **set**, adding and removing characters as we look through the input string, so the set *always only holds the characters that appear an odd number of times*.

Solution

Our approach is to check that each character appears an even number of times, allowing for only one characters to appear an odd number of times (a middle character). This is enough to determine if a permutation of the input string is a palindrome.

We iterate through each character in the input string, keeping track of the characters we've seen an odd number of times using a set `unpaired_characters`.

As we iterate through the characters in the input string:

- If the character is not in `unpaired_characters`, we add it.
- If the character is already in `unpaired_characters`, we remove it.

Finally, we just need to check if less than two characters don't have a pair.

```ruby
require 'set'

def has_palindrome_permutation(the_string)

    # track characters we've seen an odd number of times
    unpaired_characters = Set.new

    (0...the_string.length).each do |i|
        char = the_string[i]

        if unpaired_characters.include? char
            unpaired_characters.delete(char)
        else
            unpaired_characters.add(char)
        end
    end

    # the string has a palindrome permutation if it
    # has one or zero characters without a pair
    return unpaired_characters.length <= 1
end
```

Complexity

$O(n)$ time, since we're making one iteration through the n characters in the string.

Our `unpaired_characters` set is the only thing taking up non-constant space. We *could* say our space cost is $O(n)$ as well, since the set of unique characters is less than or equal to n. But we can also look at it this way: there are only so many different characters. How many? The ASCII character set has just 128 different characters (standard english letters and punctuation), while Unicode has 110,000 (supporting several languages and some icons/symbols). We might want to treat our number of possible characters in our character set as another variable k and say our space complexity is $O(k)$. Or we might want to just treat it as a constant, and say our space complexity is $O(1)$.

What We Learned

One of the tricks was to use a hash (/concept/hash-map) or set.

This is the *most common way* to get from a brute force approach to something more efficient. Especially for easier problems.

I even know interviewers who *just want to hear you say* "hash", and once they hear that they'll say, "Great, let's move on."

So always ask yourself, right from the start: "Can I save time by using a hash?"

Want more examples of hashes unlocking the optimal answer for a coding interview question? Check out these other questions (/concept/hash-map#related_questions).
