

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
- kayak
- racecar

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 finds a solution by trying *all* possible answers and picking the best one.

Say you're a cashier and need to give someone 67 cents (US) using as few coins as possible. How would you do it?

You could try running through all potential coin combinations and pick the one that adds to 67 cents using the fewest coins. That's a *brute force* algorithm, since you're trying *all* possible ways to make change.

Here are a few other brute force algorithms:

- Trying to fit as many overlapping meetings as possible in a conference room? Run through all possible schedules, and pick the schedule that fits the most meetings in the room.

- Trying to find the cheapest route through a set of cities? Try all possible routes and pick the cheapest one.
- Looking for a minimum spanning tree in a graph (/concept/graph)? Try all possible sets of edges, and pick the cheapest set that's also a tree.

Brute force solutions are usually *very slow* since they involve testing a huge number of possible answers.

Brute force approaches are rarely the most efficient. Other approaches, like greedy algorithms (/concept/greedy) or dynamic programming (/concept/bottom-up) tend to be faster.

Even so, talking through a brute force solution can be a good first step in a coding interview. It's usually pretty easy to derive, so it allows you to quickly make progress and come up with *something* that works. From there, you have some helpful boundaries for refining your algorithm—you're only interested in solutions that are faster (and/or more space efficient) than the brute force solution you've already come up with.

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 dictionary!

A **hash table** organizes data so you can quickly look up values for a given key.

Strengths:

- **Fast lookups.** Lookups take $O(1)$ time *on average*.
- **Flexible keys.** Most data types can be used for keys, as long as they're hashable (/concept/hashing).

Weaknesses:

- **Slow worst-case lookups.** Lookups take $O(n)$ time *in the worst case*.

	Average	Worst Case
space	$O(n)$	$O(n)$
insert	$O(1)$	$O(n)$
lookup	$O(1)$	$O(n)$
delete	$O(1)$	$O(n)$

- **Unordered.** Keys aren't stored in a special order. If you're looking for the smallest key, the largest key, or all the keys in a range, you'll need to look through every key to find it.
- **Single-directional lookups.** While you can look up the *value* for a given key in $O(1)$ time, looking up the *keys* for a given *value* requires looping through the whole dataset— $O(n)$ time.
- **Not cache-friendly.** Many hash table implementations use linked lists (</concept/linked-list>), which don't put data next to each other in memory.

In Python 2.7

In Python 2.7, hash tables are called dictionaries.

```
light_bulb_to_hours_of_light = {  
    'incandescent': 1200,  
    'compact fluorescent': 10000,  
    'LED': 50000,  
}
```

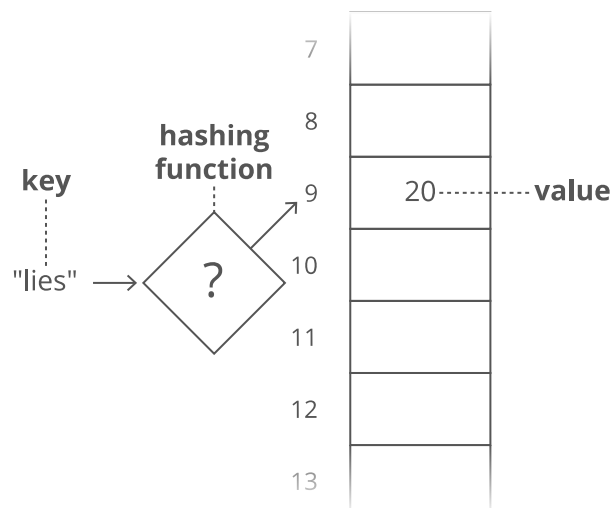
Python 2.7 ▼

Hash maps are built on arrays

Arrays (</concept/array>) are pretty similar to hash maps already. Arrays let you quickly look up the value for a given "key" . . . except the keys are called "indices," and we don't get to pick them—they're always sequential integers (0, 1, 2, 3, etc).

Think of a hash map as a "hack" on top of an array to let us use flexible keys instead of being stuck with sequential integer "indices."

All we need is a function to convert a key into an array index (an integer). That function is called a **hashing function** (</concept/hashing>).



To look up the value for a given key, we just run the key through our hashing function to get the index to go to in our underlying array to grab the value.

How does that hashing function work? There are a few different approaches, and they can get pretty complicated. But here's a simple proof of concept:

Grab the number value for each character and add those up.

$$\begin{array}{c}
 \text{" l i e s "} \\
 \downarrow \downarrow \downarrow \downarrow \\
 108 + 105 + 101 + 115 = 429
 \end{array}$$

The result is 429. But what if we only have 30 slots in our array? We'll use a common trick for forcing a number into a specific range: the modulus operator (%). (/concept/modulus) Modding our sum by 30 ensures we get a whole number that's less than 30 (and at least 0):

$$429 \% 30 = 9$$

The hashing functions used in modern systems get pretty complicated—the one we used here is a simplified example.

Hash collisions

What if two keys hash to the same index in our array? In our example above, look at "lies" and "foes":

$$\begin{array}{cccc}
 \text{"} & \text{l} & \text{i} & \text{e} & \text{s} & \text{"} \\
 \downarrow & \downarrow & \downarrow & \downarrow & & \\
 108 & +105 & +101 & +115 & = & 429 \\
 & \uparrow & \uparrow & \uparrow & \uparrow & \\
 \text{"} & \text{f} & \text{o} & \text{e} & \text{s} & \text{"}
 \end{array}$$

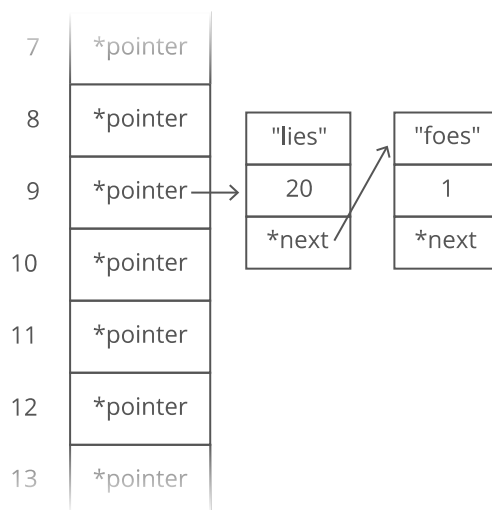
$$102 + 111 + 101 + 115 = 429$$

They both sum up to 429! So of course they'll have the same answer when we mod by 30:

$$429 \% 30 = 9$$

This is called a **hash collision**. There are a few different strategies for dealing with them.

Here's a common one: instead of storing the actual values in our array, let's have each array slot hold a *pointer* to a *linked list* (/concept/linked-list) holding the values for all the keys that hash to that index:



Notice that we included the *keys* as well as the values in each linked list node. Otherwise we wouldn't know which key was for which value!

There are other ways to deal with hash collisions. This is just one of them.

When hash table operations cost $O(n)$ time

Hash collisions

If *all* our keys caused hash collisions, we'd be at risk of having to walk through all of our values for a single lookup (in the example above, we'd have one big linked list). This is unlikely, but it *could* happen. That's the worst case.

Dynamic array resizing

Suppose we keep adding more items to our hash map. As the number of keys and values in our hash map exceeds the number of indices in the underlying array, hash collisions become inevitable.

To mitigate this, we could expand our underlying array whenever things start to get crowded. That requires allocating a larger array and rehashing all of our existing keys to figure out their new position— $O(n)$ time.

Sets

A **set** is like a hash map except it only stores keys, without values.

Sets often come up when we're tracking groups of items—nodes we've visited in a graph, characters we've seen in a string, or colors used by neighboring nodes. Usually, we're interested in whether something is in a set or not.

Sets are usually implemented very similarly to hash maps—using hashing to index into an array—but they don't have to worry about storing values alongside keys. In Python, the set implementation is largely copied from the dictionary implementation (<https://markmail.org/message/ktzomp4uwrnmnza06>).

```
light_bulbs = set()

light_bulbs.add('incandescent')
light_bulbs.add('compact fluorescent')
light_bulbs.add('LED')

'LED' in light_bulbs # True
'halogen' in light_bulbs # False
```

Python 2.7 ▼

. (**Tip:** using a dictionary 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 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 dictionary 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 character 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.

```
def has_palindrome_permutation(the_string):
    # Track characters we've seen an odd number of times
    unpaired_characters = set()

    for char in the_string:
        if char in unpaired_characters:
            unpaired_characters.remove(char)
        else:
            unpaired_characters.add(char)

    # The string has a palindrome permutation if it
    # has one or zero characters without a pair
    return len(unpaired_characters) <= 1
```

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 dictionary (/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* "dictionary", 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 dictionary?"

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

Hash Table/Hashing Coding Interview Questions

MillionGazillion »

I'm making a new search engine called MillionGazillion(tm), and I need help figuring out what data structures to use. keep reading »

<https://www.interviewcake.com/question/python/compress-url-list>

Inflight Entertainment »

Writing a simple recommendation algorithm that helps people choose which movies to watch during flights keep reading »

<https://www.interviewcake.com/question/python/inflight-entertainment>

Permutation Palindrome »

Check if any permutation of an input string is a palindrome. keep reading »

Top Scores »

Efficiently sort numbers in an array, where each number is below a certain maximum. keep reading »

<https://www.interviewcake.com/question/python/top-scores>

Word Cloud Data »

You're building a word cloud. Write a function to figure out how many times each word appears so we know how big to make each word in the cloud. keep reading »

<https://www.interviewcake.com/question/python/word-cloud>

Find Duplicate Files »

Your friend copied a bunch of your files and put them in random places around your hard drive. Write a function to undo the damage. keep reading »

<https://www.interviewcake.com/question/python/find-duplicate-files>

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.