**Strobogrammatic Number**

Given a string num which represents an integer, return true *if* num *is a****strobogrammatic number***.

A **strobogrammatic number** is a number that looks the same when rotated 180 degrees (looked at upside down).

**Example 1:**

**Input:** num = "69"

**Output:** true

**Example 2:**

**Input:** num = "88"

**Output:** true

**Example 3:**

**Input:** num = "962"

**Output:** false

**Constraints:**

* 1 <= num.length <= 50
* num consists of only digits.
* num does not contain any leading zeros except for zero itself.

/\*\*

\* @param {string} num

\* @return {boolean}

\*/

var isStrobogrammatic = function(num) {

};

Solution Article

Overview

Given a number represented as a string, we need to return true if it is strobogrammatic and false if it is not. Determining whether or not a number is strobogrammatic requires looking at whether or not it stays the same when rotated by 180 degrees.

We aren't told very much about which digits count as strobogrammatic. While some people have found it frustrating to not be told everything, this level of uncertainty is intentionally typical of real interviews. A big part of solving this problem is clarifying the requirements, and there is a great way of doing that here on LeetCode, which we'll talk about.

**Rotating a number by 180 degrees**

A good first step is to think carefully about what it means to rotate a number by 180 degrees. Some people find it easy to rotate numbers in their head, whereas others struggle to do so. If you're in the latter group, write some numbers down on a scrap of paper and rotate the paper.

1 / 10

Observe that rotating a number by 180 degrees reverses the order of the digits and rotates each digit upside-down in its new position. It isn't guaranteed that the rotation will form a valid number, though; some digits will become invalid when rotated in this way.

**Investigating the rotation of each digit**

The next step is to determine what each digit becomes when rotated by 180 degrees. There are three possibilities for each digit:

1. it becomes invalid
2. it stays the same
3. it becomes a different digit

We'll consider a *digit* to be **rotatable** if, and only if, that digit becomes a valid digit when rotated. For example, 9 becomes 6, and 8 remains as 8. On the other hand, 4 becomes ߈, which is clearly no longer valid. So, we can say that 6, 9, and 8 are rotatable, whereas 4 is *not*. Numbers such as 1 might be rotatable, but it is font dependent.

So, which digits are rotatable? Your interviewer is unlikely to just give you the answer; you'll probably be expected to figure this out by asking good questions and stating your assumptions and conclusions. The main challenge is that the rotatable-ness of a number is somewhat dependent on the font being used.

Remember, somewhat vague requirements are standard in interviews, and the first step you should take in any interview is to analyze the question, ensure you understood it, clarify the requirements, and state your assumptions. The interviewer *expects* you to ask questions; they don’t want you to *guess* the requirements!

A good idea would be to quickly go through each of the ten digits in order, giving an initial thought on whether or not the digit is rotatable (it's okay if your initial thoughts are different from mine; we're not committed to this yet).

* 0 is probably considered rotatable. Most people draw it as a circle without a "decoration" in the middle.
* 1 may be considered rotatable. On one hand, it's commonly drawn as simply a vertical bar, but on the other, sometimes decorations are added, and most fonts put it off-center.
* 2 may be considered rotatable. It looks a lot like itself when rotated, although there is a distinctive difference we need to be wary of (see the pictures above).
* 3 is *not* rotatable.
* 4 is *not* rotatable.
* 5 may be considered rotatable, for the same reason 2 might be.
* 6 is rotatable; we were shown that it becomes 9.
* 7 is *not* rotatable.
* 8 is probably considered rotatable. Most people draw the two "circles" the same size.
* 9 is rotatable; we were shown it becomes 6.

The next step is dependent on how your interviewer responds to your initial thoughts. They might give you a more precise definition, ask you to make your own judgment call, or they might even ask you to come up with some specific test cases whose answers would clarify the definition. We're going to go with this latter possibility, as it's what you should do on LeetCode.

Before you read any further, have a think about which test cases you could use.

The general rule of thumb for interviews is that you should *state* your assumptions (and be ready to justify them if asked) and *ask* clarifying questions for anything you’re unsure about. Keep in mind that your interviewer is more likely to give you useful answers to specific questions, as opposed to general questions. For example, asking them to list all 1 and 2 digit strobogrammatic numbers might seem like it would be very helpful to you, but they probably won’t be keen on doing your work for you. Instead, you should, for example, ask whether or not 1 and 22 are strobogrammatic numbers. You might need to briefly explain why the cases are unclear (e.g., font).

Here are the simplest test cases you could use. We're going to assume that 3, 4, and 7 are *not* rotatable.

* 0: Will return true if 0 is rotatable.
* 1: Will return true if 1 is rotatable.
* 8: Will return true if 8 is rotatable.
* 2: Will return true if 2 is rotatable.
* 5: Will return true if 5 is rotatable.
* 69: Will return true if 6 and 9 are rotatable and become each other (we already know this from the example given in the problem, but you might not be given it in an interview).

**On LeetCode, you can simulate the clarification process using the Run Code button!**. Note that this process will **not** affect your solution acceptance statistics. You can use the *Run Code* button as much as you want. If you're not familiar with the trick, here's how you would check whether or not 2 is considered to be strobogrammatic.

1. If you haven’t yet written any code, you’ll first need to modify the template function to return a dummy value (e.g. return false).
2. Type 2 into the *Testcase* panel.
3. Click the *Run Code* button.
4. Check what the ***Expected*** output is. In this case, it is false, allowing us to conclude that 2 is **not** a strobogrammatic number.

After checking test cases, we can refine our thoughts from above.

8, 0, and 1 are themselves upside-down. 6 and 9 are each other upside-down. All other numbers, including 2 and 5, are not themselves up-side-down, nor are they any other number up-side-down.

Therefore, the number 68199866189 is rotatable.

1 / 10

But the number 9619196 is not.

1 / 10

Learning to deal with vague requirements in a calm and methodical way is an often overlooked but *very* important soft skill for both interviews and day-to-day work as a software engineer. If you fail an interview due to misunderstanding the requirements, then this is a sign that you need to work on your requirement clarification skills and *not* a sign that the interviewer just asked you a "stupid" question.

From here, we're going to look at a couple of different ways of solving this problem.

Approach 1: Make a Rotated Copy

**Intuition**

A straightforward solution is to make a copy of the input that is rotated by 180 degrees. If the rotated copy is identical to the original input, then the input has to be strobogrammatic.

1 / 13

**Algorithm**

Recall that when a number is rotated by 180 degrees, *the order of the digits reverses* and *each digit is rotated upside-down in its new position*. As such, we could build a new rotated string by looping through the original string backward (to reverse it) and rotating + appending each digit to the new string.

Recall from above that the rules for rotating a character are as follows:

* 0 ⟶ 0
* 1 ⟶ 1
* 6 ⟶ 9
* 8 ⟶ 8
* 9 ⟶ 6

The digits 2, 3, 4, 5, and 7 are not rotatable. Their presence immediately signifies that the input number couldn't possibly be rotatable, and therefore not strobogrammatic.

The simplest way of doing the rotations in code is to use if statements.

define function isStrobogrammatic(num\_string):

rotated\_string = an empty string

for each character in num\_string, going in reverse:

if character is 0, 1, or 8:

append character to rotated\_string

else if character is 6:

append 9 to rotated\_string

else if character is 9:

append 6 to rotated\_string

else (character is invalid):

return false

if rotated\_string is the same as num\_string:

return true

return false

Alternatively, you could use a hash map, or even an array, to store the rules for flipping. I've provided code for these alternate approaches in the next section.

**Code**

Recall that we should always use a **string builder** to build a new string, and *not* repeated string concatenations.

In most programming languages, the built-in string type is *immutable*. This means that if the string is modified, then a new string object must be made. Appending N*N* characters to a string would, therefore, have a cost of O(N^2)*O*(*N*2), as for each of the N*N* characters, a new string of length N / 2*N*/2 (on average) has to be created. This problem is solved by using string builders. A string builder is a list-like data structure that characters are inserted into. Once all characters have been inserted, the characters in the string builder are converted to a string object.

Here is the code that uses if statements (based on the algorithm from above). Catching the "invalid digit" case in the else is best, as it avoids the need for five statements within a single condition.

class Solution {

public boolean isStrobogrammatic(String num) {

// Note that using a String here and appending to it would be

// poor programming practice.

StringBuilder rotatedStringBuilder = new StringBuilder();

// Remember that we want to loop backwards through the string

for (int i = num.length() - 1; i >= 0; i--) {

char c = num.charAt(i);

if (c == '0' || c == '1' || c == '8') {

rotatedStringBuilder.append(c);

} else if (c == '6') {

rotatedStringBuilder.append('9');

} else if (c == '9') {

rotatedStringBuilder.append('6');

} else { // This must be an invalid digit.

return false;

}

}

String rotatedString = rotatedStringBuilder.toString();

return num.equals(rotatedString);

}

}

Here is the code using a Hash Map to avoid the need for a complex conditional statement.

class Solution {

public boolean isStrobogrammatic(String num) {

// Initialise a map with the five-digit rotation rules

Map<Character, Character> rotatedDigits = new HashMap<> (

Map.of('0', '0', '1', '1', '6', '9', '8', '8', '9', '6'));

StringBuilder rotatedStringBuilder = new StringBuilder();

// Remember that we want to loop backwards through the string

for (int i = num.length() - 1; i >= 0; i--) {

char c = num.charAt(i);

if (!rotatedDigits.containsKey(c)) {

return false; // This must be an invalid digit.

}

rotatedStringBuilder.append(rotatedDigits.get(c));

}

String rotatedString = rotatedStringBuilder.toString();

return num.equals(rotatedString);

}

}

Alternatively, we could use an Array instead of a Hash Map; the indexes act as keys. It is simplest in code to map the non-rotatable characters to empty strings instead of explicitly checking for them during the string building process. If they were present, then the rotated string will be of a different length to the original, and therefore would be correctly flagged as not strobogrammatic in the final check. You could also check for the invalid characters in the same way we did for the hash map approach

This approach is nice in that the code is very compact—the conditional inside the loop has been eliminated—but not so nice in that it is somewhat confusing to understand. The Hash Map approach is probably safer in an interview.

class Solution {

public boolean isStrobogrammatic(String num) {

// In Java, we need to put '\0' to represent an empty character

char[] rotatedDigits = new char[]{'0', '1', '\0', '\0', '\0', '\0', '9', '\0', '8', '6'};

StringBuilder rotatedStringBuilder = new StringBuilder();

// Remember that we want to loop backwards through the string

for (int i = num.length() - 1; i >= 0; i--) {

char c = num.charAt(i);

int charIndex = Character.getNumericValue(c);

rotatedStringBuilder.append(rotatedDigits[charIndex]);

}

String rotatedString = rotatedStringBuilder.toString();

return num.equals(rotatedString);

}

}

**Complexity Analysis**

Let N*N* be the length of the input string.

* Time complexity : O(N)*O*(*N*).

In the worst case, all digits in the string will be rotatable. We're going to assume this for the time complexity analysis.

For each of the N*N* digits in the string, we're looking up (with a hash map, array, or cascading if) the rotation of that digit. For all three sub-implementations, this has a cost of O(1)*O*(1). Appending to the end of a string builder is also O(1)*O*(1). Therefore, building the 180-degree rotation of a string has a cost of O(N)*O*(*N*).

In the final step, we're comparing two strings of length N*N*. This also has a cost of O(N)*O*(*N*).

This gives us O(N) + O(N)*O*(*N*)+*O*(*N*). In Big-O notation, we treat this as simply O(N)*O*(*N*).

* Space complexity : O(N)*O*(*N*).

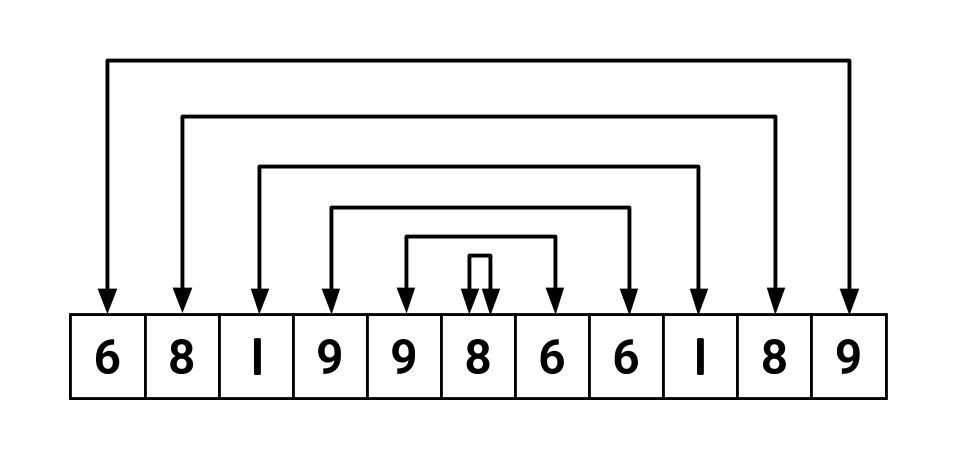
The string builder requires O(N)*O*(*N*) space.

Note that if you didn't implement your string building sensibly (e.g., if you used string concatenation), then your solution will have a time complexity of O(N^2)*O*(*N*2).

Approach 2: Two Pointers

**Intuition**

You might have observed a pattern in how digits *move* in the rotation: the first and last swap, the second and the second-to-last swap, etc.



For the number to be strobogrammatic, we have to write the **same** number back into each index. As we deduced earlier, there are only five valid pairs of numbers for this to work.

1. 0 and 0.
2. 1 and 1.
3. 6 and 9.
4. 8 and 8.
5. 9 and 6.

Therefore, we can check each pair that would swap in the reversal for whether or not it is one of the five pairs listed above. If all pairs are on the list, then the number is strobogrammatic. For odd-lengthed numbers, the middle digit has to be 0, 1, or 8.

1 / 6

**Algorithm**

We initialize two pointers; left and right. We then iterate both pointers towards the middle at each step, ensuring that the digits at left and right correspond to one of the five valid pairs. An elegant way of doing this is to define a hash map of valid left -> right mappings (like what we did in the hash map variant of approach 1).

define function isStrobogrammatic(num):

rotations = a new hash map

add to rotations: '0' -> '0', '1' -> '1', '8' -> '8', '6' -> '9', and '9' -> '6'

left = 0

right = num.length - 1

while left <= right:

if left is not in rotations:

return false

expected\_rotation = get rotation for character at position left in num

if expected\_rotation is not the same as the character at right in num:

return false

add 1 to left

subtract 1 from right

return true

If no invalid pairs are found, then the number must be strobogrammatic. Note that the middle-digit-of-an-odd-number case is handled correctly; the final iteration will have left = right. If they are both pointing to the same 0, 1, or 8, then the condition will be false, and true returned at the end. If they are both pointing at a 6, then the condition will be true, and false will be returned, as expected\_rotation will be 9, and num[right] will be 6.

Be careful of that middle value in an odd-lengthed number. While the number 8 9 0 6 8 is strobogrammatic, the number 8 9 4 6 8 is not. And nor is 8 9 9 6 8 (this last example is one that is particularly likely to catch some people out, as the middle digit is rotatable, but it doesn't become itself).

**Code**

class Solution {

public boolean isStrobogrammatic(String num) {

Map<Character, Character> rotatedDigits = new HashMap<> (

Map.of('0', '0', '1', '1', '6', '9', '8', '8', '9', '6'));

// Java allows us to have more than one iteration variable.

for (int left = 0, right = num.length() - 1; left <= right; left++, right--) {

char leftChar = num.charAt(left);

char rightChar = num.charAt(right);

if (!rotatedDigits.containsKey(leftChar) || rotatedDigits.get(leftChar) != rightChar) {

return false;

}

}

return true;

}

}

**Complexity Analysis**

Let N*N* be the length of the input string.

* Time complexity : O(N)*O*(*N*).

For each of the N*N* digits in the string, we're doing a single lookup and comparison.

* Space complexity : O(1)*O*(1).

We are only using constant extra space. This is an in-place algorithm.