1056. Confusing Number

## **Problem Description**

Easy

A confusing number is defined as an integer which, when rotated by 180 degrees, yields a different number, but still maintains its validity by only consisting of valid digits. Each digit has its rotated counterpart as follows:

- 0, 1, 8 remain unchanged when rotated  $(0 \rightarrow 0, 1 \rightarrow 1, 8 \rightarrow 8)$ . • 6 and 9 are swapped when rotated  $(6 \rightarrow 9, 9 \rightarrow 6)$ . • Digits 2, 3, 4, 5, and 7 do not have valid rotations and thus make a number invalid if present after rotation.
- When a number is rotated, we disregard leading zeros. For example, 8000 becomes 0008 after rotation, which we treat as 8.
- The task is to determine whether a given integer n is a confusing number. If n is a confusing number, the function should return

true; otherwise, it returns false.

## To solve this problem, we need to check two things:

Intuition

- 2. Whether the rotated number is different from the original number.
- To start with, we map each digit to its rotated counterpart (if any), with invalid digits being mapped to -1. This gives us the array

d with precomputed rotated values for all possible single digits:

after rotation and building the rotated number at the same time.

(e.g., 2, 3, 4, 5, or 7), its rotated value in the list is -1.

simultaneously performs integer division and modulo operation.

value, y is first multiplied by 10 and then the rotated digit is added to it.

the rotated number in-place using basic arithmetic operations.

 $\circ$  We use the list d = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6].

1. Whether each digit in the given number has a valid rotation.

[0, 1, -1, -1, -1, -1, 9, -1, 8, 6]The intuition for the solution is to iterate through the digits of n from right to left, checking that each digit has a valid rotated

```
counterpart, and simultaneously building the rotated number. This is achieved by the following steps:
```

1. Initialize two variables, x and y. x will hold the original number which we'll deconstruct digit by digit, and y will be used to construct the rotated number. 2. We use a loop to process each digit of x until all digits have been processed:

- $\circ$  x, v = divmod(x, 10) uses Python's divmod function to get the last digit v and update x to remove this last digit. ∘ We then check if the current digit v has a valid rotation by looking it up in the array d. If d[v] is -1, we have an invalid digit; in this case, we
- If v is valid, we compute the new rotated digit and add it to y by shifting y to the left (by a factor of 10) and then adding d[v]. 3. After processing all digits of x, we end up with y, which is the number formed after rotating n. We compare y with n to check if they are different.
- If they are the same, it means the number is not confusing and we return false. Otherwise, we return true. Solution Approach
- The implementation uses a simple algorithm that involves iterating through the digits of the given number to check for validity

## Initialize variables: The solution starts with initializing two variables, x and y. x is assigned the value of the given number n

Here's the breakdown:

return false.

and will be used to iterate through its digits. y is initialized to 0 and will be used to construct the rotated number. Predefined rotations: A list d is created that defines the rotation of each digit. This list serves as a direct mapping, where the

index represents the original digit and the value at that index represents the rotated digit. If a digit is invalid when rotated

- Iterate through digits: The while-loop is used to iterate through the digits of n from right to left. Inside the loop, divmod(x, 10) obtains the rightmost digit v of x and updates x to eliminate the rightmost digit. divmod is a Python built-in function that
- is invalid upon rotation, and the function returns false. An example is if the original number contains a 2, since 2 does not have a valid rotation equivalent. Building rotated number: If the digit is valid, the rotated digit (found at d[v]) is added to y. To maintain the correct place

Validity check: The solution then checks for the validity of each digit by referencing the d list. If d[v] is -1, it means the digit v

**Final check**: After processing the entire number, y would now be the rotated number. The rotated number y is then compared

confusing number and the function returns false. Otherwise, it returns true. This algorithm efficiently checks each digit of the number without the need for additional data structures and effectively builds

with the original number n. If they are identical, it means that the rotation has not changed the number, hence it is not a

**Example Walkthrough** Let's use the number 619 as a small example to illustrate the solution approach. Initialize variables:

 x is assigned the value of n, so x becomes 619. y is initialized to 0. Predefined rotations:

## Validity check and building rotated number:

Final check:

number.

number.

**Python** 

class Solution:

Iterate through digits:

• First Iteration (rightmost digit 9): • x, v = divmod(619, 10) gives x = 61, v = 9.

d[v] = d[9] is 6 (valid rotation), so we move to building y.

619 has three digits, so we will perform the process below three times, once for each digit.

• x, v = divmod(61, 10) gives x = 6, v = 1. • d[v] = d[1] is 1, 1 remains the same after rotation.

Second Iteration (middle digit 1):

y = y \* 10 + d[v] = 0 \* 10 + 6 = 6.

y = y \* 10 + d[v] = 6 \* 10 + 1 = 61.

y = y \* 10 + d[v] = 61 \* 10 + 9 = 619.

• Third Iteration (leftmost digit 6):  $\blacksquare$  x, v = divmod(6, 10) gives x = 0, v = 6 (as x is now less than 10).

d[v] = d[6] is 9 (valid rotation).

 $\circ$  Now x is 0, and we have finished processing the digits. We have y = 619.

3. x, v = divmod(6, 10) gives x = 0, v = 6.

 $\circ$  d[6] is 9, so y = 8 \* 10 + 9 = 89.

- We compare y with the original n, and in this case, 619 is equal to 619, meaning the number did not change upon rotation.
- 1. Initialize x = 68 and y = 0. 2. x, v = divmod(68, 10) gives x = 6, v = 8.  $\circ$  d[8] is 8, so y = 0 \* 10 + 8 = 8.

Determine if the given number is a confusing number. A confusing number is a number that,

or the number remains the same after rotation, it is not a confusing number.

# Obtain the last digit and reduce the original number by one digit

# Check for valid rotation, return False if rotation is invalid (indicated by -1)

original\_number, last\_digit = divmod(original\_number, 10)

# Build the rotated number by appending the rotated digit

when rotated 180 degrees, becomes a different valid number. If any digit cannot be rotated,

To see how a confusing number would work with this example, let's rotate the number 68:

Since the rotated number is identical to the original number, 619 is not a confusing number according to our definition.

Therefore, the function would return false for 619 because rotating the number gives us the same number instead of a different

Solution Implementation

4. The original number 68 is different from the rotated number 89, therefore the function would return true, indicating that 68 is indeed a confusing

```
# Original number and transformed/rotated number
original_number = n
rotated_number = 0
```

# Mapping of digits after 180-degree rotation

# Process each digit of the original number

if rotation\_map[last\_digit] < 0:</pre>

// Function to check if a number is a confusing number

// Digit mapping, with -1 representing invalid mappings

int transformedNumber = 0; // Initialize the transformed number

int digit = originalNumber % 10; // Get the last digit

return false; // This is not a confusing number

transformedNumber = transformedNumber \* 10 + map[digit];

// Remove the last digit from the original number for next iteration

// Build the transformed number by adding the mapped digit at the appropriate place

// A number is confusing if it's not equal to the original number after transformation

vector<int> map =  $\{0, 1, -1, -1, -1, -1, 9, -1, 8, 6\};$ 

// Process each digit to create the transformed number

// Digit is not valid if it cannot be mapped

int originalNumber = n; // Store the original number

while original\_number:

return False

rotation\_map = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6]

:param n: The input number to be tested.

:return: True if n is a confusing number, False otherwise.

def confusingNumber(self, n: int) -> bool:

```
rotated_number = rotated_number * 10 + rotation_map[last_digit]
       # A number is confusing if it is different from its rotation
       return rotated_number != n
Java
class Solution {
   // Method to determine if a number is a confusing number
    public boolean confusingNumber(int n) {
       // Mappings from original digit to its possible flipped digit
       // -1 indicates an invalid digit that doesn't have a valid transformation
       int[] digitTransformations = new int[] \{0, 1, -1, -1, -1, -1, 9, -1, 8, 6\};
       // Original number
       int originalNumber = n;
       // Transformed number after flipping the digits
       int transformedNumber = 0;
       // Process each digit of the original number
       while (originalNumber > 0) {
            // Get the last digit of the current number
            int digit = originalNumber % 10;
            // Check if the digit has a valid transformation
            if (digitTransformations[digit] < 0) {</pre>
               // If not, it's not a confusing number
               return false;
           // Update the transformed number with the flipped digit
            transformedNumber = transformedNumber * 10 + digitTransformations[digit];
           // Remove the last digit from the original number
           originalNumber /= 10;
       // The number is confusing if the transformed number is different from the original number
       return transformedNumber != n;
C++
```

```
// Digit mapping, with undefined representing invalid mappings
const digitMap: (number | undefined)[] = [0, 1, undefined, undefined, undefined, undefined, 9, undefined, 8, 6];
```

**TypeScript** 

**}**;

class Solution {

bool confusingNumber(int n) {

while (originalNumber) {

if (map[digit] < 0) {</pre>

originalNumber /= 10;

return transformedNumber != n;

public:

```
// Function to check if a number is a confusing number
  function confusingNumber(n: number): boolean {
      let originalNumber: number = n; // Store the original number
      let transformedNumber: number = 0; // Initialize the transformed number
      // Process each digit to create the transformed number
      while (originalNumber > 0) {
          const digit: number = originalNumber % 10; // Get the last digit
          // The digit is not valid if it cannot be mapped (i.e., it is undefined in digitMap)
          if (digitMap[digit] === undefined) {
              return false; // This is not a confusing number
          // Build the transformed number by adding the mapped digit at the appropriate place
          transformedNumber = transformedNumber * 10 + digitMap[digit]!;
          // Remove the last digit from the original number for the next iteration
          originalNumber = Math.floor(originalNumber / 10);
      // A number is confusing if it's not equal to the original number after transformation
      return transformedNumber !== n;
class Solution:
   def confusingNumber(self, n: int) -> bool:
       Determine if the given number is a confusing number. A confusing number is a number that,
       when rotated 180 degrees, becomes a different valid number. If any digit cannot be rotated,
       or the number remains the same after rotation, it is not a confusing number.
        :param n: The input number to be tested.
        :return: True if n is a confusing number, False otherwise.
       # Original number and transformed/rotated number
       original_number = n
        rotated_number = 0
       # Mapping of digits after 180-degree rotation
        rotation_map = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6]
       # Process each digit of the original number
       while original_number:
           # Obtain the last digit and reduce the original number by one digit
            original_number, last_digit = divmod(original_number, 10)
            # Check for valid rotation, return False if rotation is invalid (indicated by -1)
            if rotation_map[last_digit] < 0:</pre>
               return False
```

Time and Space Complexity The time complexity of the given code is  $0(\log n)$ , where n is the input number. This complexity arises because the code

processes each digit of the number exactly once, and there are O(log n) digits in a base-10 number. The space complexity of the code is 0(1) since it uses a constant amount of extra space regardless of the input size. The variables x, y, and v along with the array d are the only allocations, and their size does not scale with n.

# Build the rotated number by appending the rotated digit

# A number is confusing if it is different from its rotation

return rotated\_number != n

rotated\_number = rotated\_number \* 10 + rotation\_map[last\_digit]