<https://leetcode.com/problems/fraction-to-recurring-decimal/description/>

Given two integers representing the numerator and denominator of a fraction, return **the fraction in string format**.

If the fractional part is repeating, enclose the repeating part in parentheses.

If multiple answers are possible, return **any of them**.

It is **guaranteed** that the length of the answer string is less than 104 for all the given inputs.

**Example 1:**

**Input:** numerator = 1, denominator = 2

**Output:** "0.5"

**Example 2:**

**Input:** numerator = 2, denominator = 1

**Output:** "2"

**Example 3:**

**Input:** numerator = 4, denominator = 333

**Output:** "0.(012)"

**Constraints:**

-2^31 <= numerator, denominator <= 2^31 - 1

denominator != 0

**Attempt 1: 2023-02-08**

**Solution 1: Hash Table + Math (60 min)**

class Solution {

    public String fractionToDecimal(int numerator, int denominator) {

        // Handle of 'numerator = 0', test out by:

        // Input: numerator = 0, denominator = 3

        // Output: "-0", Expected: "0"

        if(numerator == 0) {

            return "0";

        }

        StringBuilder sb = new StringBuilder();

        boolean isNegative = (numerator > 0) ^ (denominator > 0);

        sb.append(isNegative ? "-" : "");

        // Handle range out of integer, test out by:

        // Input: numerator = -1, denominator = -2147483648

        // Output: "0.0000000000000000000000000000001",

        // Expected: "0.0000000004656612873077392578125"

        // ------------------------------------------------

        // Handle of negative number in either 'numerator' or 'denominator', test out by:

        // Input: numerator = -50, denominator = 8

        // Output: "--6.-2-5", Expected: "-6.25"

        // Input: numerator = 7, denominator = -12

        // Output: "-0.-5-8(-3)", Expected: "-0.58(3)"

        // We have to separately Math.abs() on 'numerator' and 'denominator', only

        // Math.abs() on 'quotient' or 'remainder' won't work, because if 'denominator'

        // is negative, still error out on test case 2 above

        long num = Math.abs((long)numerator);

        long denom = Math.abs((long)denominator);

        long quotient = num / denom;

        sb.append(quotient);

        long remainder =  num % denom;

        if(remainder == 0) {

            return sb.toString();

        }

        sb.append(".");

        /\*

                  0.01201201...

            333 / 4 00000000...

                  3 33

                    670

                    666

                    400

                    333

                        670

                        666

                        400

                        333

                        ......

         \*/

        // Map to store already seen remainders and their positions in StringBuilder

        Map<Long, Integer> map = new HashMap<>();

        // Loop until no remainder or repeated remainder is found

        while(remainder != 0) {

            map.put(remainder, sb.length());

            // Multiply by 10 to get the numerator for next digit

            remainder \*= 10;

            sb.append(remainder / denom);

            remainder %= denom;

            if(map.containsKey(remainder)) {

                sb.insert(map.get(remainder), "(");

                sb.append(")");

                break;

            }

        }

        return sb.toString();

    }

}

Time Complexity: O(N)

Space Compleixty: O(N)

**Refer to**

<https://algo.monster/liteproblems/166>

**Problem Description**

The problem requires us to represent a fraction given by two integers as a string. The numerator is the top part of the fraction, while the denominator is the bottom part. The output should be a non-reduced fraction converted into a decimal format string. If the decimal part of the fraction repeats indefinitely, we need to identify this repeating sequence and enclose it within parentheses in our string.

To clarify with an example, if we are given numerator = 1 and denominator = 3, the decimal part of this fraction is 0.333..., which repeats indefinitely. Therefore, the output should be "0.(3)" to indicate that the 3 is a repeating sequence.

Also, it's important to handle the sign of the result correctly. If the fraction is negative (meaning the numerator and denominator have opposite signs), there should be a minus sign before the fraction.

**Intuition**

The intuition behind the solution can be broken down into a few steps:

Handle Zero: If the numerator is 0, the result is 0, regardless of the denominator.

Determine the Sign: Check the sign of the numerator and denominator. If only one of them is negative, the result should have a minus sign.

Integer Part: Divide the absolute values of the numerator by the denominator to get the integer part of the result. Add the integer part to the result string.

Fractional Part: After getting the integer part, we need to process the fractional part (if there is any). Take the remainder of the division (this would be our new numerator) and proceed to the next step.

Recurring Decimals: To represent the fraction part, we multiply the remainder by 10 and then divide by the denominator to get each decimal digit. We continue this process and keep track of the remainders in a map/dictionary to detect repetition. If the remainder repeats, that means we have found a cycle and thus a recurring decimal. We use this information to place parentheses in the string around the repeating part.

Constructing the Fraction String: We keep appending the results of the division to the answer string. Once the repetition is detected, we insert a '(' at the correct position (using the map/dictionary to find where the cycle started) and add a ')' at the end of the result string.

This solution leverages long division and a mapping strategy to successfully note and handle recurring decimals, constructing the desired string representation for the fraction.

**Solution Approach**

The implementation of the solution can be broken down into the following key algorithmic steps:

Identifying whether the result is negative:

This is done by checking if the numerator and the denominator have opposite signs using an exclusive OR operation numerator > 0) ^ (denominator > 0). If the result is negative, a negative sign '-' is added in the beginning.

Extracting integer part:

The absolute values of the numerator and denominator are found using abs(), and the integer part of the fraction is obtained by integer division num // d. This part is added to the result string as the initial part before the decimal point.

Handling the fractional part:

The remainder num % d after the integer division becomes the new numerator for the fractional part. If the remainder is zero, the function immediately returns the result because there is no fractional part.

Setting up a map to identify repeating fractions:

A map (or dictionary in Python), named mp, is set up to remember the position of each remainder in the output string. The position is stored so that when we encounter the same remainder again, we know that the sequence of decimals is starting to repeat.

Performing the long division for the decimal part:

Long division is simulated by multiplying the remainder (new numerator) by 10 and again performing integer division by the denominator to find the next digit of the decimal part. This process (num \*= 10) continues in a loop until we either reach a remainder of zero or find a repeated remainder.

Detecting repeating sequences:

During each iteration of the long division, before appending the next digit to the result, we check if the current remainder is already in the map.

If it's not, we add the remainder along with the current length of the result string to the map (mp[num] = len(res)).

If it is, we've identified a repeating sequence. We insert a parenthesis at the index where this remainder first appeared (res.insert(idx, '(')), and add the closing parenthesis at the end of the current result (res.append(')')), and then break out of the loop.

Building the final result string:

The result is formed by combining all the parts of the fraction into a single string with ''.join(res).

The key data structures used in this solution are:

A list (res) to construct the string since strings in Python are immutable, and updating an immutable string repeatedly would be inefficient.

A dictionary (mp) to keep track of the remainders and their positions in the result list, which is crucial for identifying repeating decimals.

Here is an illustration of the process with numerator 1 and denominator 6 which should yield the result "0.1(6)":

Integer division gives us the integer part 0 and a remainder of 1.

We set up our result list with ['0', '.'] and our map as {}.

We multiply the numerator by 10, getting 10, and dividing by 6 gives us 1 with a remainder of 4.

We append the 1 to our result list (making it ['0', '.', '1']) and store the position of the remainder 4 in the map ({4: 3}).

We repeat the multiplication and division to get another digit which is 6 with a remainder of 4 again.

Since 4 is already in the map, we insert a ( at the index from the map (making the result list ['0', '.', '1', '(', '6']) and append a ) at the end (finally having ['0', '.', '1', '(', '6', ')']).

We join the list and return the string "0.1(6)".

The algorithm is effective at handling both non-repeating and repeating decimals, terminating appropriately once the division is complete or a repetition is found.

**Example Walkthrough**

Let's use a small example to illustrate the solution approach with the numerator 1 and the denominator 7. The expected result is "0.(142857)" since 1/7 results in a repeating decimal with repeating sequence "142857".

Now, we walk through the steps of the solution:

Identifying whether the result is negative:

Both 1 and 7 are positive, so the result will also be positive. No negative sign is needed.

Extracting integer part:

Integer division of 1 // 7 equals 0, since 1 is less than 7. The result string starts with "0.".

Handling the fractional part:

The remainder now is 1 % 7 which is 1, so there is a fractional part. This remainder is the starting point for our long division.

Setting up a map to identify repeating fractions:

We create an empty map mp = {} to keep track of our remainders and where they first appear in the result.

Performing the long division for the decimal part:

We multiply the remainder by 10, getting 10, and divide by 7 which gives us 1 with a new remainder 3. We append 1 to our result string and add 3 to the map pointing to its current index in the result string, index 2 (right after "0.").

Detecting repeating sequences:

We repeat the multiplication and division. With the new remainder 3, we get 30 / 7 which is 4 with remainder 2. We do this process iteratively:

20 / 7 gives 2 and the remainder 6

60 / 7 gives 8 and the remainder 4

40 / 7 gives 5 and the remainder 5

50 / 7 gives 7 and the remainder 1

When we reach the remainder 1 again, it's already in our map; it indicates the start of the repeating sequence. Thus, we insert a parenthesis at the index of the first occurrence of 1 in our result list and a closing parenthesis at the end of the list.

Building the final result string:

The result string now becomes "0.142857". To represent the repeating sequence, we place parentheses around "142857" to get "0.(142857)".

Throughout every step of this example, the algorithm carefully handles the integer part and fractional part of the fraction, accurately recording each digit of the repeating decimal, and correctly identifies the beginning of the repeating sequence to enclose it within parentheses, resulting in the correct representation of 1/7 as a string with repeating decimal notation.

**Java Solution**

class Solution {

    public String fractionToDecimal(int numerator, int denominator) {

        // Check for zero numerator which means the result is 0

        if (numerator == 0) {

            return "0";

        }

        // StringBuilder to build the final string result

        StringBuilder resultBuilder = new StringBuilder();

        // Determine the sign of the result (neg if numerator XOR denominator is negative)

        boolean isNegative = (numerator > 0) ^ (denominator > 0);

        resultBuilder.append(isNegative ? "-" : "");

        // Convert to long to prevent integer overflow

        long num = Math.abs((long) numerator);

        long denom = Math.abs((long) denominator);

        // Append the integer part of the quotient to the result string

        resultBuilder.append(num / denom);

        num %= denom;  // Get the remainder

        // If there is no remainder, we can return the result as it's not a fraction

        if (num == 0) {

            return resultBuilder.toString();

        }

        // Fraction part starts after the decimal point

        resultBuilder.append(".");

        // Map to store already seen remainders and their positions in resultBuilder

        Map<Long, Integer> remainderPositions = new HashMap<>();

        // Loop until no remainder or repeated remainder is found

        while (num != 0) {

            remainderPositions.put(num, resultBuilder.length());

            num \*= 10; // Multiply by 10 to get the numerator for next digit

            resultBuilder.append(num / denom); // Append the quotient digit

            num %= denom;  // Get the new remainder

            // Check if this remainder has been seen before

            if (remainderPositions.containsKey(num)) {

                // Insert the opening parenthesis at the position of the first occurrence of this remainder

                int index = remainderPositions.get(num);

                resultBuilder.insert(index, "(");

                // Append closing parenthesis at the end of the result string

                resultBuilder.append(")");

                break;

            }

        }

        // Convert the StringBuilder to a string and return it

        return resultBuilder.toString();

    }

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the function fractionToDecimal is primarily dependent on the length of the resulting decimal fraction. In the worst case, this length could be equal to the value of the denominator d, because the longest recurring cycle in a fraction numerator/denominator cannot have more than d - 1 digits.

Within the while loop, multiplication, division, and modulus operations are computed, which are all constant-time operations (O(1)). However, the loop can run up to d times in the worst case, implying a time complexity of O(d).

Therefore, the worst-case time complexity is O(d).

**Space Complexity**

The space complexity involves the storage used by the res list and the mp dictionary:

The res list's size grows linearly with the length of the decimal fraction, which in the worst case can be O(d).

The mp dictionary maps previously seen remainders to their corresponding index in res. In the worst case, we might store up to d - 1 distinct remainders, resulting in a space complexity of O(d) for the dictionary.

Thus, the overall space complexity of the function fractionToDecimal is also O(d).

**Refer to**

[Decimals](note://WEB2a93da8cb4699db72b00a4740496ab86)

[Long Division](note://WEB32b0e1b30a7e68957bf995a65f3d4b4d)