1447. Simplified Fractions

Medium Math String Number Theory

Problem Description

In this problem, we are given an integer `n` and we need to find all unique fractions where the numerator (top number) is less than the denominator (bottom number), the denominator is between 2 and `n`, and the fraction is in its simplest form. A fraction is in its simplest form if the greatest common divisor (GCD) of its numerator and denominator is 1, meaning the fraction cannot be reduced any further. We need to list the fractions that are in between (but not including) 0 and 1, which means the numerator will always be less than the denominator. The output should be a list of these fractions represented as strings and they can be in any order.

Intuition

To solve this problem, we use a nested loop approach where we iterate through all possible numerators (`i`) and denominators (`j`).

1. We initiate the outer loop with `i` starting from 1 to `n - 1`. The numerator must be less than the denominator, hence it is less than `n`.
2. We begin the inner loop with `j` starting from `i + 1` to `n`. The reason `j` starts at `i + 1` is to ensure that we always have a fraction (since `i` is less than `j`), and to make sure the denominator is greater than the numerator for valid fractions between 0 and 1.
3. For every combination of (`i`, `j`) we check if the GCD of `i` and `j` is 1. This check is performed using the `gcd` function. If it is 1, then the fraction (`i/j`) is in its simplest form and can be included in our list.
4. If the GCD is 1, we create a string representation of the fraction by using string formatting, so `i` and `j` are inserted into the string in the form `'{i}/{j}'`.
5. All the fractions that meet the condition are collected into a list using list comprehension and returned as the final answer.

The use of the embedded loops is suitable here because we need to compare each possible numerator with each possible denominator, and we require this comprehensive checking to ensure that we don't miss any simplified fractions.

Solution Approach

The solution to this problem follows a direct brute-force approach, by iterating over the range of possible numerators and denominators, then checking for their simplicity before including them in the results. Below is a step-by-step breakdown of the implementation strategy.

1. **List Comprehension**: The entire implementation utilizes Python's list comprehension, which is a concise way to create lists. It allows us to iterate over an iterable, apply a filter or condition, and process the items, all in a single, readable line.
2. **Nested Loops**: We make use of nested loops within the list comprehension. The outer loop iterates over all possible numerators `i` from 1 up to `n-1`. The inner loop iterates over all possible denominators `j` from `i+1` up to `n`. This ensures we only consider fractions where the numerator is less than the denominator.
3. **Greatest Common Divisor (GCD)**: To ensure that we only take the simplified fractions (irreducible fractions), we check the GCD of the numerator and denominator. The `gcd` function from Python's standard library (usually from `[math](/problems/math-basics)` module, though not explicitly shown in the code provided) calculates the GCD. If `gcd(i, j)` equals 1, it means `i` and `j` are coprime and thus the fraction `i/j` is already in its simplest form.
4. **String Formatting**: For each fraction that is in simplified form, we format the numerator and denominator into a string of the form `'i/j'`. This is done using an f-string `f'{i}/{j}'`, which is a way to embed expressions inside string literals using curly braces.
5. **Output**: The output is a list of these formatted string representations of all simplified fractions for the given `n`.

No additional data structures or complex patterns are used here. The simplicity of this solution is its strength, as it directly answers the need without unnecessary complications, making full use of Python's built-in functionalities to handle the problem in an efficient and understandable manner.

Example Walkthrough

Let's illustrate the solution approach using a small example with `n = 5`.

1. We start with an empty list to hold our results.
2. The outer loop will run with `i` starting from 1 to 4 (since `i` ranges from 1 to `n - 1`, which is `5 - 1`).
3. For each `i`, the inner loop will run with `j` starting from `i + 1`, up to 5 (`n`).

We will go through iterations like this:

- For `i = 1`:
  - `j` will range from 2 to 5. So we will check the fractions `1/2`, `1/3`, `1/4`, and `1/5`.
  - All these fractions are in their simplest forms, so all of them will be added to the results list.
- For `i = 2`:
  - `j` will range from 3 to 5. We will check the fractions `2/3`, `2/4`, and `2/5`.
  - `2/3` and `2/5` are in their simplest forms, so they will be added.
  - `2/4` can be reduced to `1/2` (which is already included), hence, it will not be added.
- For `i = 3`:
  - `j` will range from 4 to 5. We check `3/4` and `3/5`.
  - Both fractions are in their simplest forms and will be added to the results list.
- For `i = 4`:
  - `j` is 5. We check `4/5`.
  - `4/5` is already in its simplest form, so it gets added.

Finally, the results list would consist of all the fractions in their simplest forms within the range `1/i` to `4/5`, which are: `1/2`, `1/3`, `1/4`, `1/5`, `2/3`, `2/5`, `3/4`, and `4/5`.

As assured by the solution approach, no fractions are repeated, and all are in the simplest form. This approach leverages the power of list comprehension paired with nested loops and GCD checks in Python to effectively gather the results.

Solution Implementation

Python

```python
from typing import List
from math import gcd

class Solution:
    def simplifiedFractions(self, n: int) -> List[str]:
        # List comprehension to generate simplified fractions
        return [
            f"{numerator}/{denominator}"  # format as a fraction
            for numerator in range(1, n)          # loop over numerators
            for denominator in range(numerator + 1, n + 1)  # loop over denominators
            if gcd(numerator, denominator) == 1  # include fraction if gcd is 1 i.e., fractions are simplified
        ]

# Example of how to use the class:
# sol = Solution()
# print(sol.simplifiedFractions(4))  # Output: ['1/2', '1/3', '1/4', '2/3', '3/4']
```

Java

```java
class Solution {
    // Generates a list of simplified fractions less than 1 for a given integer n
    public List<String> simplifiedFractions(int n) {
        // List to hold the result of simplified fractions
        List<String> fractions = new ArrayList<>();

        // Loop over all possible numerators
        for (int numerator = 1; numerator < n; ++numerator) {
            // Loop over all possible denominators greater than the numerator
            for (int denominator = numerator + 1; denominator <= n; ++denominator) {
                // Add the fraction to the list if the numerator and denominator are coprime
                if (gcd(numerator, denominator) == 1) {
                    fractions.add(numerator + "/" + denominator);
                }
            }
        }
        return fractions;
    }

    // Helper method to calculate the greatest common divisor (GCD) using Euclid's algorithm
    private int gcd(int a, int b) {
        // Recursively compute gcd, with the base case being when b is 0
        return b == 0 ? gcd(b, a % b) : a;
    }
}
```

C++

```cpp
#include <vector>
#include <string>
#include <algorithm> // for std::gcd
using namespace std;

class Solution {
public:
    // Generates a list of simplified fractions for denominators up to 'maxDenominator'
    vector<string> simplifiedFractions(int maxDenominator) {
        vector<string> simplifiedFractionsList; // Stores the list of simplified fractions

        // Iterate over all possible numerators
        for (int numerator = 1; numerator < maxDenominator; ++numerator) {
            // Iterate over all possible denominators
            for (int denominator = numerator + 1; denominator <= maxDenominator; ++denominator) {
                // Check if the fraction is in simplest form (gcd is 1)
                if (gcd(numerator, denominator) == 1) {
                    // Concatenate the parts of the fraction into a string
                    simplifiedFractionsList.push_back(to_string(numerator) + "/" + to_string(denominator));
                }
            }
        }
        // Return the list of simplified fractions
        return simplifiedFractionsList;
    }
};
```

TypeScript

```typescript
/**
 * Returns an array of simplified fractions up to a given limit.
 * @param {number} maxDenominator - The maximum denominator to consider for the fractions.
 * @returns {string[]} - An array of strings representing simplified proper fractions.
 */
function simplifiedFractions(maxDenominator: number): string[] {
    // Initialize an array to hold the simplified fractions.
    const fractions: string[] = [];
    // Loop over all possible numerators.
    for (let numerator = 1; numerator < maxDenominator; ++numerator) {
        // Loop over all possible denominators greater than the numerator.
        for (let denominator = numerator + 1; denominator <= maxDenominator; ++denominator) {
            // Check if the fraction is already in simplest form.
            if (gcd(numerator, denominator) === 1) {
                // If so, add the fraction to the array.
                fractions.push(`${numerator}/${denominator}`);
            }
        }
    }
    // Return the array of simplified fractions.
    return fractions;
}

/**
 * Calculates the greatest common divisor (GCD) of two numbers using the Euclidean algorithm.
 * @param {number} a - The first number.
 * @param {number} b - The second number.
 * @returns {number} - The GCD of the two numbers.
 */
function gcd(a: number, b: number): number {
    // Base case: if the second number is 0, the GCD is the first number.
    return b === 0 ? a : gcd(b, a % b);
}
```

```python
from typing import List
from math import gcd

class Solution:
    def simplifiedFractions(self, n: int) -> List[str]:
        # List comprehension to generate simplified fractions
        return [
            f"{numerator}/{denominator}"  # format as a fraction
            for numerator in range(1, n)          # loop over numerators
            for denominator in range(numerator + 1, n + 1)  # loop over denominators
            if gcd(numerator, denominator) == 1  # include fraction if gcd is 1 i.e., fractions are simplified
        ]

# Example of how to use the class:
# sol = Solution()
# print(sol.simplifiedFractions(4))  # Output: ['1/2', '1/3', '1/4', '2/3', '3/4']
```

Time and Space Complexity

Time Complexity

The time complexity of the code is calculated by considering the nested for-loops and the call to `gcd` (Greatest Common Divisor) function for each combination.

- The outer loop runs from `1` to `n - 1` which gives us `n - 1` iterations.
- The inner loop runs from `i + 1` to `n` for each value of `i` from the outer loop. In the worst case, the inner loop runs `n/2` times on average (since starting point increases as `i` increases).
- The `gcd` function can be assumed to run in `O(log(min(i, j)))` in the worst case, which is generally due to the Euclidean algorithm for computing the greatest common divisor.

Therefore, the time complexity is approximately `O(n^2 * log(n))`. However, this is an upper bound; the average case might be lower due to early termination of the `gcd` calculations for most pairs.

Space Complexity

The space complexity is determined by the space required to store the output list. In the worst case scenario, the number of simplified fractions is maximized when `n` is a prime number, resulting in a list of size `1/2 * (n - 1) * (n - 2)` (since each number from `1` to `n - 1` will be paired with every number greater than it up to `n`).

Thus, the space complexity of the solution is `O(n^2)`.