1276. Number of Burgers with No Waste of Ingredients

Medium Math

Problem Description

The problem provides us with two integers, `tomatoSlices` and `cheeseSlices`, representing the number of tomato and cheese slices available to make burgers. There are two types of burgers: Jumbo Burgers, which require `4` tomato slices and `1` cheese slice, and Small Burgers, which require `2` tomato slices and `1` cheese slice.

We are asked to find how many of each type of burger we can make such that after making them, there are no remaining tomato or cheese slices. If it is possible, we return a list with two elements, `[total_jumbo, total_small]`, representing the count of Jumbo and Small Burgers, respectively. If it's not possible to use all the tomato and cheese slices exactly, we return an empty list `[]`.

Intuition

To solve this problem, we can set up a system of linear equations based on the constraints given for making Jumbo and Small Burgers:

1. Each Jumbo Burger requires `4` tomato slices and `1` cheese slice.
2. Each Small Burger requires `2` tomato slices and `1` cheese slice.

Let `x` be the number of Jumbo Burgers and `y` be the number of Small Burgers. We can derive two equations from the problem statement:

- `4x + 2y = tomatoSlices`: The total tomatoes used has to equal the available tomato slices.
- `x + y = cheeseSlices`: The total cheese used has to equal the available cheese slices.

We want to find non-negative integer solutions for `x` and `y` that satisfy both equations simultaneously. Rearranging the second equation gives us `x = cheeseSlices - y`. Substituting this in the first equation allows us to solve for `y`:

```
4(cheeseSlices - y) + 2y = tomatoSlices
4cheeseSlices - 4y + 2y = tomatoSlices
4cheeseSlices - tomatoSlices = 2y
y = (4cheeseSlices - tomatoSlices) / 2
```

This gives us a direct way to calculate the number of Small Burgers `y`, and subsequently, we can find `x` (number of Jumbo Burgers). However, `y` must be a non-negative integer for the solution to be valid. Therefore, `4cheeseSlices - tomatoSlices` must be a non-negative even number to make `y` a non-negative integer. Similarly, `x` must also be a non-negative integer.

The provided solution in Python reflects this logic. It ensures that the value for `y` (`k // 2`) is not fractional (by checking if `k % 2` is not zero) and both `x` and `y` are not negative before returning `[x, y]`. If any of these conditions is not met, it returns `[]`, signifying that there is no possible solution.

Solution Approach

The implementation of the solution adopts a direct approach to solving the system of linear equations derived from the problem constraints.

The primary algorithm used here is to derive two linear equations from the given constraints and simplify them to express one variable in terms of the other. The equations derived are as follows:

1. The total tomatoes used for Jumbo and Small Burgers: `4x + 2y = tomatoSlices`
2. The total cheese used for Jumbo and Small Burgers: `x + y = cheeseSlices`

These two equations can be simplified to find `y`:

1. Multiply the second equation by 2: `2(x + y) = 2 * cheeseSlices`
2. Subtract the modified second equation from the first: `(4x + 2y) - (2x + 2y) = tomatoSlices - 2 * cheeseSlices`
3. Simplify to find `y`: `2x + tomatoSlices - 2 * cheeseSlices` and hence `x = (tomatoSlices - 2 * cheeseSlices) / 2`

Now that we have `x` in terms of `tomatoSlices` and `cheeseSlices`, we can find `y` by substituting `x` back into one of the original equations:

```
y = cheeseSlices - x
```

The `Solution` class's `numOfBurgers` method takes the inputs `tomatoSlices` and `cheeseSlices` and performs these operations to find `x` and `y`. Here are the steps in the code:

1. `k = 4 * cheeseSlices - tomatoSlices`: This is to isolate the `y` term by combining and rearranging the original equations.
2. `y = k // 2`: Integer division by `2` to solve for `y`.
3. `x = cheeseSlices - y`: Substitute `y` back into the `x + y = cheeseSlices` equation to solve for `x`.

Before returning `[x, y]`, the method checks three conditions:

- `k % 2`: This checks if `k` is an even number, as `y` must be a whole number.
- `y < 0`: This checks if `y` is non-negative.
- `x < 0`: This checks if `x` is non-negative.

If any of these conditions fail, it indicates that there are no non-negative integer solutions to the problem with the given `tomatoSlices` and `cheeseSlices`, and the method returns an empty list `[]`.

Otherwise, it means valid counts for Jumbo (`x`) and Small (`y`) Burgers have been found that use all the `tomatoSlices` and `cheeseSlices`, and the solution `[x, y]` is returned.

The solution utilizes basic arithmetic operations and integer division, which are efficient operations in terms of computational complexity.

Example Walkthrough

Let's consider a small example with `8` tomato slices and `5` cheese slices.

We want to find out if it's possible to make some combination of Jumbo (4 tomatoes, 1 cheese) and Small (2 tomatoes, 1 cheese) burgers that exactly use up all the tomato and cheese slices.

First, we use the given relationships to create our equations:

- For tomatoes: `4x + 2y = 8` (Equation 1)
- For cheese: `x + y = 5` (Equation 2)

Now, let's solve for `y` using these equations.

Step 1: Rearrange Equation 2 to isolate `x`: `x = 5 - y`

Step 2: Substitute `x` in Equation 1 with `5 - y`:

```
4(5 - y) + 2y = 8
20 - 4y + 2y = 8
20 - 8 = 2y
12 = 2y
```

Step 3: Solve for `y`: `y = 12 / 2`, so `y = 6`.

However, since `y` (the number of Small Burgers) is `6` and `x + y` must be `5`, we already know something is wrong because we have more Small Burgers than the total number of cheese slices, which is not possible.

As the calculated `y` exceeds the number of cheese slices, we would subtract `y` from the total cheese to find `x`: `x = 5 - 6`, which gives us `x = -1`. We cannot have a negative number of burgers.

Since we have found impossible values (negative or more than available cheese slices) for `x` and `y`, we conclude that there is no solution that exactly uses up all `8` tomato slices and `5` cheese slices with the given constraints.

In correspondence with the Solution class in Python, the method `numOfBurgers(8, 5)` would return an empty list `[]`, indicating no solution.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def numOfBurgers(self, tomato_slices: int, cheese_slices: int) -> List[int]:
        # Find the difference between quadruple the cheese slices and the tomato slices
        difference = 4 * cheese_slices - tomato_slices

        # Calculate the number of 'small' burgers (with 2 tomato slices each)
        num_small_burgers = difference // 2

        # Calculate the number of 'jumbo' burgers (with 4 tomato slices each)
        num_jumbo_burgers = cheese_slices - num_small_burgers

        # The result should be valid only if the difference is even, and neither of burgers is negative
        if difference % 2 == 0 and num_small_burgers >= 0 and num_jumbo_burgers >= 0:
            return [num_jumbo_burgers, num_small_burgers]
        else:
            # If any of the conditions above are not met, return an empty list
            return []

# Example usage:
# solution = Solution()
# print(solution.numOfBurgers(16, 7)) # Should output [1, 6] since it is possible to make 1 jumbo and 6 small burgers
```

Java

```java
class Solution {

    // Method to calculate the number of Jumbo and Small burgers that can be made given
    // the number of tomato and cheese slices.
    public List<Integer> numOfBurgers(int tomatoSlices, int cheeseSlices) {
        // Calculate the difference between four times the number of cheese slices and tomato slices
        int difference = 4 * cheeseSlices - tomatoSlices;

        // Calculate the number of Small burgers by dividing the difference by 2
        int numSmallBurgers = difference / 2;

        // Calculate the number of Jumbo burgers by subtracting the number of Small burgers
        // from the total cheese slices
        int numJumboBurgers = cheeseSlices - numSmallBurgers;

        // Check if difference is even and both calculated burger amounts are non-negative
        boolean isSolutionValid = difference % 2 == 0 && numSmallBurgers >= 0 && numJumboBurgers >= 0;

        // Return the number of Jumbo and Small burgers, if possible; otherwise, return an empty list
        return isSolutionValid ? Arrays.asList(numJumboBurgers, numSmallBurgers) : Collections.emptyList();
    }
}
```

C++

```cpp
class Solution {
public:
    // This method calculates the number of jumbo and small burgers one can make
    // with the given number of tomato and cheese slices.
    // Jumbo burgers require 4 tomato slices and 1 cheese slice.
    // Small burgers require 2 tomato slices and 1 cheese slice.
    vector<int> numOfBurgers(int tomatoSlices, int cheeseSlices) {
        // Calculate excess tomato slices after subtracting 4 times the cheese slices.
        // This should be even and non-negative for a valid combination.
        int excessTomato = 4 * cheeseSlices - tomatoSlices;

        // If excessTomato is even, half of it represents the difference in number of
        // tomato slices used by small burgers compared to jumbo burgers.
        int smallBurgers = excessTomato / 2;

        // Subtracting the number of small burgers from total cheese slices gives
        // us the number of jumbo burgers.
        int jumboBurgers = cheeseSlices - smallBurgers;

        // If excessTomato is odd or if we end up with negative counts for either burger,
        // return an empty vector (no solution).
        return (excessTomato % 2 == 0 && jumboBurgers >= 0 && smallBurgers >= 0) ?
               vector<int>{jumboBurgers, smallBurgers} : vector<int>{};
    }
};
```

TypeScript

```typescript
// This function calculates the number of jumbo and small burgers one can make
// with the given number of tomatoSlices and cheeseSlices.
// Jumbo burgers require 4 tomato slices and 1 cheese slice.
// Small burgers require 2 tomato slices and 1 cheese slice.
// It returns an array with the number of jumbo and small burgers one can make,
// or an empty array if there's no valid combination.
function numOfBurgers(tomatoSlices: number, cheeseSlices: number): number[] {
    // Calculate excess tomato slices after subtracting 2 times the number of cheese slices.
    // This excess must be even and non-negative for a valid combination.
    const excessTomato: number = tomatoSlices - 2 * cheeseSlices;

    // If the excess is divisible by 2, it represents the additional tomatoes
    // needed to upgrade small burgers to jumbo burgers.
    const conversionToJumbo: number = excessTomato / 2;

    // The number of jumbo burgers is equal to the excess divided by 2,
    // since each conversion to a jumbo requires 2 additional tomatoes.
    const jumboBurgers: number = cheeseSlices - conversionToJumbo;

    // If the excess is odd or we have a negative count of either type of burger,
    // return an empty array to indicate no solution is possible.
    // Otherwise, return the counts in an array where the first element is jumbo burgers
    // and the second is small burgers.
    if (excessTomato % 2 === 0 && jumboBurgers >= 0 && conversionToJumbo >= 0) {
        return [jumboBurgers, conversionToJumbo];
    } else {
        return [];
    }
}
```

```python
from typing import List

class Solution:
    def numOfBurgers(self, tomato_slices: int, cheese_slices: int) -> List[int]:
        # Find the difference between quadruple the cheese slices and the tomato slices
        difference = 4 * cheese_slices - tomato_slices

        # Calculate the number of 'small' burgers (with 2 tomato slices each)
        num_small_burgers = difference // 2

        # Calculate the number of 'jumbo' burgers (with 4 tomato slices each)
        num_jumbo_burgers = cheese_slices - num_small_burgers

        # The result should be valid only if the difference is even, and neither of burgers is negative
        if difference % 2 == 0 and num_small_burgers >= 0 and num_jumbo_burgers >= 0:
            return [num_jumbo_burgers, num_small_burgers]
        else:
            # If any of the conditions above are not met, return an empty list
            return []

# Example usage:
# solution = Solution()
# print(solution.numOfBurgers(16, 7)) # Should output [1, 6] since it is possible to make 1 jumbo and 6 small burgers
```

Time and Space Complexity

Time Complexity

The given code consists of simple arithmetic calculations and conditional checks, which do not depend on the size of the input but are executed a constant number of times. Therefore, the time complexity is $O(1)$.

Space Complexity

The space complexity of the code is also $O(1)$ since it uses a fixed amount of space for the variables `k`, `y`, `x`, and the return list regardless of the input size. The solution does not utilize any additional data structures that grow with the size of the input.