13. Roman to Integer

Easy | Hash Table | Math | String

Problem Description

Roman numerals are a numeral system from ancient Rome based on combinations of letters from the Latin alphabet (I, V, X, L, C, D, M) to represent numbers. Each symbol has a fixed numerical value, with I equal to 1, V equal to 5, X equal to 10, L equal to 50, C equal to 100, D equal to 500, and M equal to 1000. Numbers in Roman numerals are formed by combining these symbols and adding up their values. However, when a smaller numeral comes before a larger one, this indicates that the smaller numeral should be subtracted from the larger numeral instead of added (for example, IV means 4, not 6). This rule is used in six instances: I before V and X, X before L and C, and C before D and M. The task is to convert a string representing a Roman numeral into the integer it represents.

Intuition

To solve the problem, we should first create a mapping of Roman numeral symbols to their respective integer values. This allows for easy lookup during the conversion. To convert the Roman numeral to an integer, we can iterate over the string from left to right, checking the value of each symbol in comparison to the symbol to its right.

If a symbol is followed by one of greater value, it means we need to subtract the value of the current symbol from our result. Otherwise, we can simply add the value of the symbol. This adheres to the subtraction rule given for cases like IV or IX. To implement this in Python, we can take advantage of the pairwise utility from the `itertools` module, which will give us each symbol and the one following it. If the module `itertools` is not available or if `pairwise` is not a part of `itertools`, we can use a simple 'zip' technique to iterate through the symbol pairs. Then, we just need to add the value of the last symbol at the end since it's not included in these pairwise comparisons.

The algorithm is relatively straightforward: we initialize a sum to 0 and iterate the symbols pairwise (current and next symbol). If the current is less than the next one, we subtract its value from the sum. If the current is greater than or equal to the next one, we add its value to the sum. After the loop, we catch the edge case by including the last symbol's value, since it's always added to the total.

Solution Approach

The solution involves a straightforward process which utilizes a hash table and a simulation algorithm based on the rules of Roman numerals.

1. We define a hash table (in Python, a dictionary) `d` which maps each Roman numeral symbol to its integer value. The key-value pairs in this dictionary are as follows: `'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000`.
2. To convert a Roman numeral string to an integer, we iterate over the string one character at a time, examining the symbol and the one that follows it (this is the pairwise comparison). For this task, we use a for loop in conjunction with the Python generator expression format. To demonstrate, we generate tuples of (current character `a`, next character `b`) using the pairwise utility or a manual method by zipping the string with itself offset by one character.
3. In this iteration, we compare the integer value of the current symbol with the value of the symbol following it. If the current value is less than the next, we should deduct its value from the total sum since it indicates subtraction as per Roman numerals (e.g., `IV` for 4). If the current value is equal to or greater than the following symbol's value, we add it to the total sum.
4. The comparison `(d[a] < d[b])` returns a boolean, and in the solution, it is used to determine whether the value will be subtracted `-1 * d[a]` or added `1 * d[a]`. We use a sum function to add all these values together.
5. As the pairwise comparison does not include the very last symbol in the computation, we correct for this by adding `d[s[-1]]` to the total sum to include the value of the last symbol.
6. The final sum that results from this process is the integer value of the provided Roman numeral string.

To demonstrate the implementation of our approach with the code:

```
class Solution:
    def romanToInt(self, s: str) -> int:
        d = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
        total = sum((-1 if d[a] < d[b] else 1) * d[a] for a, b in pairwise(s)) + d[s[-1]]
        return total
```

In the above code, `pairwise(s)` would need to be replaced by an equivalent if it's not available in the standard Python library: `zip(s, s[1:])`. Here, `zip` pairs each character with the next, effectively creating the needed pairwise functionality.

```
# If pairwise is not included, we can define our own pairwise function like this:
from itertools import tee

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

# Alternatively, we can modify the loop in the main function to not need pairwise:
class Solution:
    def romanToInt(self, s: str) -> int:
        d = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
        total, prev_value = 0, 0
        for char in reversed(s):
            if d[char] < prev_value:
                total -= d[char]
            else:
                total += d[char]
            prev_value = d[char]
        return total
```

By adapting the algorithm to avoid using the `pairwise` utility, we ensure compatibility with the Python standard library without relying on any external or updated modules.

Example Walkthrough

Let's use the Roman numeral `MCMIV` as an example to illustrate the solution approach, which translates to `1904` in integer form. This number is chosen because it includes subtraction instances (`CM` and `IV`).

Following the solution approach steps:

1. We create a dictionary called `d` that maps each Roman numeral to its integer value:

   ```
   d = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
   ```

2. We start iterating over the Roman numeral string "MCMIV" using pairs:

   - Pair MC (M=1000, C=100)
   - Pair CM (C=100, M=1000)
   - Pair MI (M=1000, I=1)
   - Pair IV (I=1, V=5)

3. We compare the values of each pair:

   - For MC, 1000 is greater than 100, so we add 1000.
   - For CM, 100 is less than 1000, so we subtract 100.
   - For MI, 1000 is greater than 1, so we add 1000.
   - For IV, 1 is less than 5, so we subtract 1.

4. We perform the addition and subtraction as directed:

   - Starting sum is 0.
   - Add M (1000) = 1000.
   - Subtract C (100) from sum (900).
   - Add M (1000) to sum (1900).
   - Subtract I (1) from sum (1899).

5. We add the last character value (V=5) to the sum since the pairwise comparison above doesn't account for it:

   - The final sum is 1899 + 5 = 1904, which is the integer equivalent of "MCMIV".

This methodically breaks down the input string, using the subtraction rule as needed, and continues to accumulate the total to reach the correct conversion from Roman numeral to integer.

Python Solution

```
class Solution:
    def romanToInt(self, s: str) -> int:
        # Create a dictionary mapping Roman numerals to integers.
        roman_to_int = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}

        # Initialize the previous number with the value of the last Roman numeral.
        previous_number = roman_to_int[s[-1]]

        # Initialize the total with the value of the last Roman numeral.
        total = previous_number

        # Loop over the string of Roman numerals in reverse order (right-to-left).
        for i in range(len(s) - 2, -1, -1):
            # Get the integer value of the current Roman numeral.
            current_number = roman_to_int[s[i]]

            # If the current value is less than the previous value, we need to subtract it.
            # Otherwise, we add it.
            if current_number < previous_number:
                total -= current_number
            else:
                total += current_number

            # Update the previous number for the next iteration.
            previous_number = current_number

        # Return the computed total, which is the integer equivalent of the Roman numeral string.
        return total

# Example use:
# solution = Solution()
# result = solution.romanToInt("MCMIV")
# print(result)  # Output: 1994
```

Java Solution

```
class Solution {
    public int romanToInt(String s) {
        // A string representing the Roman numerals in increasing order.
        String romanSymbols = "IVXLCDM";
        // Corresponding values of Roman numerals as per the order in the string.
        int[] values = {1, 5, 10, 50, 100, 500, 1000};
        // Map to store Roman numerals and their values for quick access.
        Map<Character, Integer> numeralToValue = new HashMap<>();

        // Populate the map with symbol-value pairs.
        for (int i = 0; i < values.length; ++i) {
            numeralToValue.put(romanSymbols.charAt(i), values[i]);
        }

        // Length of the string containing the Roman numeral.
        int length = s.length();
        // Start with the value of the last symbol as there is nothing following it to compare.
        int totalValue = numeralToValue.get(s.charAt(length - 1));

        // Loop through the string in reverse order stopping before the first character.
        for (int i = 0; i < length - 1; ++i) {
            // Determine if the value of the current symbol is less than the one following it.
            // This helps in applying the subtractive rule of Roman numerals.
            int sign = numeralToValue.get(s.charAt(i)) < numeralToValue.get(s.charAt(i + 1)) ? -1 : 1;
            // Add or subtract the value of the current symbol to the total value.
            totalValue += sign * numeralToValue.get(s.charAt(i));
        }

        // Return the computed total value as the integer value of the input Roman numeral.
        return totalValue;
    }
}
```

C++ Solution

```
#include <unordered_map>
#include <string>

class Solution {
public:
    // Function to convert a Roman numeral to an integer.
    int romanToInt(std::string s) {
        // Map to store the Roman numerals and their corresponding integer values.
        std::unordered_map<char, int> numeralToValue{
            {'I', 1},
            {'V', 5},
            {'X', 10},
            {'L', 50},
            {'C', 100},
            {'D', 500},
            {'M', 1000},
        };

        // Start by adding the value of the last character to the answer.
        int total = numeralToValue[s.back()];

        // Iterate over the string from the start to the second-to-last character.
        for (int i = 0; i < s.size() - 1; ++i) {
            // Determine the sign of the value based on the following numeral in the sequence.
            // If the current numeral is less than the next one, it should be subtracted.
            int sign = numeralToValue[s[i]] < numeralToValue[s[i + 1]] ? -1 : 1;

            // Add the current numeral's value to the total, adjusting the sign as necessary.
            total += sign * numeralToValue[s[i]];
        }

        // Return the computed integer value of the Roman numeral string.
        return total;
    }
};
```

Typescript Solution

```
// Function to convert a Roman numeral string to an integer.
function romanToInt(s: string): number {
    // A map representing the Roman numeral characters and their integer values.
    const romanToValue: Map<string, number> = new Map([
        ['I', 1],
        ['V', 5],
        ['X', 10],
        ['L', 50],
        ['C', 100],
        ['D', 500],
        ['M', 1000],
    ]);

    // Initialize the result with the integer value of the last Roman numeral character.
    let result: number = romanToValue.get(s[s.length - 1])!;

    // Iterate over the string (excluding the last character) to calculate the total value.
    for (let i = 0; i < s.length - 1; ++i) {
        // Determine the sign: -1 if the current Roman numeral is less than the one after it, else 1.
        const sign: number = romanToValue.get(s[i])! < romanToValue.get(s[i + 1])! ? -1 : 1;

        // Add the current Roman numeral's value times the determined sign to the result.
        result += sign * romanToValue.get(s[i])!;
    }

    // Return the final integer result.
    return result;
}
```

Time and Space Complexity

The time complexity of the given code can be analyzed based on the operations performed on the input string `s`. The function iterates over pairs of adjacent characters in the string, which is done by `pairwise(s)`. Since `pairwise` essentially goes through the entire string once to create these pairs, the number of operations will be proportional to the length of the string `n`. Therefore, the time complexity is `O(n)`.

In terms of space complexity, the auxiliary space used by the algorithm is for the dictionary `d` that stores the Roman numerals and their corresponding integer values. The size of `d` is constant, as there are a fixed number of Roman numerals. However, `pairwise(s)` generates an iterator which does not store all pairs in memory at once, hence it does not add to the space complexity. Therefore, the space complexity remains constant, not dependent on the length of the input string, and can be denoted as `O(1)`, rather than `O(n)`, since `n` would imply a dependency on the size of the character set that isn't present here.