2310. Sum of Numbers With Units Digit K

Medium Greedy Math Dynamic Programming Enumeration

Problem Description

In this problem, we are given two integers, `num` and `k`. We need to construct a set of positive integers that satisfy two conditions:

1. The units digit (rightmost digit) of each integer in the set is `k`.
2. The sum of all the integers in the set is `num`.

We are asked to find the minimum possible size of such a set. If it's not possible to find any such set that satisfies the conditions, we should return `-1`. It's also important to note that a set can contain multiple instances of the same integer, and an empty set has a sum of `0`.

For example, if `num = 58` and `k = 9`, one possible set satisfying the conditions could be {9, 9, 9, 9, 9, 9, 4}, which has a sum of 58 and each integer has a 9 as its units digit. The goal is to find the smallest such set.

Intuition

To come up with a solution, we need to first understand a few points about numbers and their units digits:

- The unit digit of a sum depends solely on the unit digits of the summands.
- Since we can have multiple instances of the same number in our set, we can think of this problem as trying to reach `num` by adding a certain number of `k's` to the multiple of 10 (since the units digit of a multiple of 10 is always 0).

The solution starts with an observation that we can keep adding the number `k` until we either reach `num` or we surpass it. When we add `k` to itself, we keep the unit digit the same and increase the tens digit. The key insight is that if `num` can be expressed as a sum of integers with a unit digit `k`, then there must be a combination where the remainder when `num` is divided by 10 is `k`, or `num` is a multiple of 10.

With this in mind, we iterate over the range from 1 to `num`. For each index `i`, we check if `num - k * i` is a non-negative number and is also a multiple of 10 (to ensure the rest of the digits form a multiple of 10). If both conditions are met, `i` is the minimum number of times we have to add `k` to form part of a sum that equals `num`. Therefore, `i` is the smallest possible size of our desired set.

If after iterating through the possible values of `i` we do not find a suitable number that meets the conditions, it means that it is not possible to form `num` exclusively with numbers ending in `k`, and we return `-1`.

Solution Approach

The solution follows a straightforward brute-force approach. Since the problem requires finding the minimum number of integers required to create the sum `num`, where each integer has the unit digit `k`, the algorithm performs a simple iteration to check if a set can be constructed for each possible size incrementally.

Here is a breakdown of the implementation:

1. **Early Return for Zero:** If `num` is 0, we return 0 immediately since no number is needed to sum up to zero.
2. **Iteration Over Possible Set Sizes:**
   - We start a loop where `i` runs from 1 to `num` inclusive. This loop represents the iteration over possible sizes of the set.
   - On each iteration, we calculate `t`, which is `num - k * i`. This represents the part of the sum that is not contributed by adding numbers with unit digit `k`.
     - We use the walrus operator `:=` to assign the value to `t` while checking the condition in the same statement.
3. **Checking Validity of the Set:**
   - We check if `t` is non-negative and is also a multiple of 10. Both conditions are required for `t` to potentially be the sum of integers with the unit digit 0, which when combined with `i` instances of `k` would sum up to `num`.
   - To check if `t` is a multiple of 10, we simply verify if `t % 10 == 0`.
4. **Finding the Minimum Set Size:**
   - If a valid set is found (the conditions in step 3 are met), the current value of `i` represents the minimum size of the set needed, and `i` is returned.
5. **Return -1 if No Set Exists:**
   - If no such `i` is found after completing the loop, it indicates that it is not possible to construct such a set. Therefore, the function returns `-1`.

No complex data structures are required for this solution as it relies entirely on integer operations and a single loop. The pattern used can be identified as a brute-force search, checking all possible combinations until a valid one is found or until all options have been exhausted.

One important algorithmic insight is the leveraging of modular arithmetic to identify potential matches for the conditions set by the problem. This implementation efficiently approaches the problem with a time complexity of O(num), where `num` is the integer representing the sum we are trying to reach.

Example Walkthrough

Let's illustrate the solution approach using a small example where `num = 37` and `k = 8`.

Following the provided steps:

1. **Early Return for Zero:** Since `num` is not `0`, we do not return immediately and continue with the iterative process.
2. **Iteration Over Possible Set Sizes:**
   - We start the loop with `i=1` and go up to `i=37` (because `num = 37`). In each iteration, we compute the value `t` as `num - k * i`.
3. **Checking Validity of the Set:**
   - For each value of `i`, we want to check if `t` is a non-negative multiple of 10. In other words, `t % 10` should be equal to `0`.

Let's walk through a few iterations in our example:

- `i = 1`: `t = 37 - 8 * 1 = 29`. Since `29 % 10 ≠ 0`, the set {8} with size 1 is not enough.
- `i = 2`: `t = 37 - 8 * 2 = 21`. Again, `21 % 10 ≠ 0`, so the set {8, 8} with size 2 is not enough.
- ...
- Continuing this process, we finally reach:
- `i = 4`: `t = 37 - 8 * 4 = 5`. Since `5 % 10 ≠ 0`, even the set {8, 8, 8, 8} with size 4 does not meet the requirement.
- ...
- We reach `i = 5`: `t = 37 - 8 * 5 = 37 - 40 = -3`. This is a negative number, so it is evident that the set with multiple 8s that add up to 37 does not exist. We will never find a non-negative `t` that is a multiple of 10 because `num` itself is not a multiple of 10 and `k` does not have a unit digit that could make up the difference.

4. **Finding the Minimum Set Size:**
   - Since none of our checks for a valid set has passed, we don't have a minimum size to return.
5. **Return -1 if No Set Exists:**
   - Since the loop has finished and no non-negative multiple of 10 was found for `t`, we return `-1`.

Thus, in this example, we can't find a set consisting of positive integers with the unit digit `k` (which is 8) that sums up to `num` (which is 37). The smallest possible set does not exist, hence the return value is `-1`.

Solution Implementation

Python

```python
class Solution:
    def minimumNumbers(self, num: int, k: int) -> int:
        # If num is 0, no number needs to be added, so return 0
        if num == 0:
            return 0

        # Iterate through i from 1 up to the given number
        for i in range(1, num + 1):
            # Calculate the remaining value after subtracting i times k from num
            remaining_value = num - k * i

            # Check if remaining value is non-negative and divisible by 10
            if remaining_value >= 0 and remaining_value % 10 == 0:
                # If both conditions are met, i is the minimum count of numbers needed
                return i

        # If no such number exists that the remaining value is divisible by 10, return -1
        return -1
```

Java

```java
class Solution {

    // Method to find the minimum number of non-negative integers needed
    // where the last digit of each integer is 'k', and their sum is 'num'.
    public int minimumNumbers(int num, int k) {
        // If the sum required is 0, no numbers are needed.
        if (num == 0) {
            return 0;
        }

        // Iterate from 1 to 'num' to find the minimum count of integers needed.
        for (int i = 1; i <= num; ++i) {
            // Calculate the remaining value after subtracting 'k * i'.
            int remainder = num - k * i;

            // If the remainder is non-negative and ends with a 0,
            // it means we found a valid group of 'i' numbers that sum up to 'num'.
            if (remainder >= 0 && remainder % 10 == 0) {
                return i; // Return the count 'i' as the result.
            }
        }

        // If no valid combination is found, return -1.
        return -1;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the minimum number of integers needed whose last digit is 'k'
    // that add up to the number 'num'
    int minimumNumbers(int num, int k) {
        // If num is 0, no numbers are needed, so return 0.
        if (num == 0) {
            return 0;
        }

        // Iterate over possible counts of numbers with last digit 'k'
        for (int count = 1; count <= num; ++count) {
            // Calculate the remaining value after subtracting 'count' numbers
            // with last digit 'k'
            int remainder = num - k * count;

            // If the remainder is non-negative and is a multiple of 10,
            // it means the number can be formed by 'count' numbers ending with 'k'
            if (remainder >= 0 && remainder % 10 == 0) {
                return count; // Return the count as the answer
            }
        }

        // If it's not possible to sum to 'num' with integers ending in 'k', return -1
        return -1;
    }
};
```

TypeScript

```typescript
// This function finds the minimum number of integers needed to sum up to a given number 'num',
// where each integer must end with the digit 'k'.
function minimumNumbers(num: number, k: number): number {
    // If 'num' is 0, no numbers are needed so return 0.
    if (num === 0) {
        return 0;
    }

    // Determine the last digit of 'num'.
    let lastDigit = num % 10;

    // Check every possible integer with the last digit 'k'.
    for (let i = 1; i <= 10; i++) {
        let currentNumber = i * k;
        // The maximum number of integers with the last digit 'k' that we would need is 10,
        // as every digit from 0 to 9 will have appeared with 'k' at least once.
        // If 'currentNumber' is less than or equal to 'num' and ends with the same digit as 'num',
        // it's possible to sum up to 'num' with 'i' numbers ending with 'k'.
        if (currentNumber <= num && currentNumber % 10 === lastDigit) {
            return i;
        }
    }

    // If no such number could be found within the range, return -1 to indicate failure.
    return -1;
}
```


```python
class Solution:
    def minimumNumbers(self, num: int, k: int) -> int:
        # If num is 0, no number needs to be added, so return 0
        if num == 0:
            return 0

        # Iterate through i from 1 up to the given number
        for i in range(1, num + 1):
            # Calculate the remaining value after subtracting i times k from num
            remaining_value = num - k * i

            # Check if remaining value is non-negative and divisible by 10
            if remaining_value >= 0 and remaining_value % 10 == 0:
                # If both conditions are met, i is the minimum count of numbers needed
                return i

        # If no such number exists that the remaining value is divisible by 10, return -1
        return -1
```


Time and Space Complexity

The time complexity of the given code is `O(num)` because the loop runs a maximum of `num` iterations. In each iteration, it performs constant-time operations such as arithmetic subtraction, multiplication, and modulus operation. Therefore, the upper bound of loop iterations directly scales with the value of `num`.

The space complexity of the code is `O(1)` because the space used does not depend on the size of the input `num`. The algorithm uses a fixed number of variables (`i` and `t`) that do not scale with input size.