1999. Smallest Greater Multiple Made of Two Digits

Medium Math Enumeration

Leetcode Link

Problem Description

The problem asks us to find the smallest integer greater than a given integer `k` which is also a multiple of `k`, and composed solely of one or two different digits, `digit1` and `digit2`. The result of our search must not exceed the maximum value for a signed 32-bit integer (`2**31 - 1`), and if no such number exists or if it exceeds this limit, the function should return `-1`.

The task can be summarised as finding the next multiple of `k` that adheres to the digit constraints.

Intuition

In this approach to solve the problem, we employ a breadth-first search (BFS) algorithm. This is evident from the use of a queue (`deque`) where we store the current numbers under consideration. We start the queue with an initial value of 0 and iterate, expanding each number by one level, which corresponds to adding one more digit to the end of the current number, using either `digit1` or `digit2`.

The key here is that BFS will ensure that the first number we encounter that is greater than `k` and a multiple of `k` will be the smallest, because we explore numbers level by level (from smaller to greater).

The use of the queue enables us to explore all possible combinations of `digit1` and `digit2` in an orderly fashion. We start with 0 and in each iteration, for every number `x` in the queue, we generate two new numbers - one by appending `digit1` to `x`, and the other by appending `digit2` to `x` (if `digit1` and `digit2` are not the same).

We have some checks in place as well:

1. If `digit1` and `digit2` are both 0, it's impossible to form a number greater than `k`, hence we return `-1` immediately.
2. We swap `digit1` and `digit2` if `digit1` is greater than `digit2` for consistency; however, this does not affect the outcome due to the nature of search we perform.
3. As we generate new numbers, we check two conditions before enqueuing them - whether they exceed the 32-bit signed integer limit, and whether they are greater than `k` and also a multiple of `k`. If a number passes these conditions, it's the answer we're looking for, and we return it.

By iterating over all possible combinations, the algorithm ensures that the smallest satisfying number is found or determines that no such number exists within the bounds of a 32-bit signed integer.

Solution Approach

The solution uses a Breadth-First Search (BFS) approach that leverages a queue, implemented in Python using `collections.deque`. The BFS pattern is excellent for finding the shortest path to a certain condition, which, in this case, is the smallest integer that satisfies the given criteria.

Here's the breakdown of the implementation:

1. First, we check for an edge case: if both `digit1` and `digit2` are `0`, there's no possible number that can be larger than `k` and composed only of those digits, so we return `-1`.
2. There is an optimization step where if `digit1` is greater than `digit2`, the function calls itself with the digits swapped. This ensures consistency in the BFS exploration, but fundamentally does not change the outcome since both digits will be explored.
3. We initialize a queue, `q`, with a starting number `0`. This `0` is a placeholder that represents the starting point of our search.
4. We enter a while-loop that continues indefinitely since we don't know how many iterations will be necessary to find the answer. This loop will break when we either find the solution or exceed the maximum limit for 32-bit integers.
   - We use `q.popleft()` to get and remove the current number, `x`, from the front of the queue.
   - The first condition checks if `x` exceeds `2**31 - 1`. If so, we return `-1` because we're looking for a signed 32-bit integer and have exceeded the limit.
   - The second condition checks if `x` is greater than `k` and a multiple of `k`. If `x` satisfies both conditions, `x` is the answer, and we return it.
   - If neither condition is met, we generate new numbers to explore. We do this by taking `x`, multiplying it by `10` (which effectively appends a '0' to `x`), then adding `digit1` to create a new number, and we add this number to the end of the queue.
   - If `digit1` is not equal to `digit2`, we perform another similar operation for `digit2`, thus branching our search into two paths at each step.

By queuing up all numbers formed by the available digits and systematically examining them in increasing order, we guarantee that the first appropriate number found will also be the smallest one possible. Remember that this method counts on BFS's inherent nature to explore options breadth-wise (i.e., in layers of distance from the starting point) before moving to deeper layers.

Example Walkthrough

Let's use a small example to illustrate the solution approach outlined in the content provided. Consider the following input:

- `k = 13`
- `digit1 = 5`
- `digit2 = 2`

Our goal is to find the smallest integer greater than `k` (13) that is a multiple of `k` and composed only of the digits `5` and/or `2`. Let's walk through the steps of the BFS approach:

1. Since neither `digit1` nor `digit2` is `0`, we don't have to return `-1` immediately.
2. Next, we check if `digit1` is greater than `digit2`. If so, we would swap them to keep consistency in the BFS exploration, but in this example, `digit1` (5) is already less than `digit2` (2), so no swapping is necessary.
3. We initialize our queue with a starting number of `0` and proceed with the BFS search.
4. Begin processing the queue:
   - Pop `0` from the queue (BFS starts with 0 as a placeholder).
   - Since `0` is less than `2**31 - 1` and not greater than `x`, we proceed.
   - Append `digit1` (5) to `0` by multiplying `0` by `10` and adding `5`, resulting in `5`. Enqueue `5`.
   - Append `digit2` to `0` (since `digit1` != `digit2`), resulting in `2`, and enqueue `2`.
5. Continue the BFS:
   - Pop `5` from the queue. It is not greater than `k` nor a multiple of `k`. Enqueue `55` and `52`.
   - Pop `2` from the queue. It is not greater than `k` nor a multiple of `k`. Enqueue `25` and `22`.
6. The process continues until we find a number that meets the conditions:
   - Pop `55`, `52`, `25`, and `22` one by one, but none is greater than `k` and a multiple of `k`.
   - Proceed to `555`, `552`, `525`, `522`, `255`, `252`, `225`, and `222`.
7. Eventually, we would end up popping `255` from the queue, which is greater than `13` and is a multiple of `13` (255 = 13 * 19.6153... rounded to 19 * 20).
8. As soon as we find `255`, which meets the criteria (greater than `k`, a multiple of `k`, and composed only of `digit1` and `digit2`), we return this number as our result.

In summary, the BFS algorithm explored the candidates in the following sequence: `0`, `5`, `2`, `55`, `52`, `25`, `22`, `555`, `552`, `525`, `522`, `255`, `252`, `225`, `222`, ... and so on, until it found the number `255`, which is the smallest integer satisfying all the conditions.

Python Solution

```python
from collections import deque

class Solution:
    def findInteger(self, k: int, digit_one: int, digit_two: int) -> int:
        # If both digits are 0, there is no valid solution since
        # no number greater than k can be formed using only zeros.
        if digit_one == 0 and digit_two == 0:
            return -1

        # If digit_one is greater than digit_two, swap them to simplify
        # the process of forming numbers (smaller digit comes first).
        if digit_one > digit_two:
            return self.findInteger(k, digit_two, digit_one)

        # Initialize a queue to store intermediate numbers.
        queue = deque([0])

        # Use BFS to find the smallest number greater than k that is a
        # multiple of k constructed using only the given digits.
        while True:
            current_num = queue.popleft()

            # Early exit condition if number exceeds 32-bit integer range.
            if current_num > 2**31 - 1:
                return -1

            # Check if the current number is greater than k and a multiple of k.
            if current_num > k and current_num % k == 0:
                return current_num

            # Append new numbers formed by adding each digit to the right.
            queue.append(current_num * 10 + digit_one)

            # Only append the number formed with the second digit if it is different.
            if digit_one != digit_two:
                queue.append(current_num * 10 + digit_two)

# Example usage:
sol = Solution()
# print(solution.findInteger(3, 1, 5))  # This would find the smallest number greater than 3 that
                                        # is a multiple of 3 and only consists of the digits 2 and 5.
```

Java Solution

```java
class Solution {
    public int findInteger(int k, int digit1, int digit2) {
        // If both digits are 0, no valid number > k can be composed; return -1.
        if (digit1 == 0 && digit2 == 0) {
            return -1;
        }

        // If digit1 is greater than digit2, swap them to have them in ascending order.
        if (digit1 > digit2) {
            return findInteger(k, digit2, digit1);
        }

        // Use a deque to perform a breadth-first search (BFS).
        Deque<Long> queue = new ArrayDeque<>();

        // Start with zero as the initial candidate in the queue.
        queue.offer(0L);

        // Continue the search until we find a valid number or exceed Integer.MAX_VALUE.
        while (true) {
            // Take the first candidate from the queue.
            long currentNumber = queue.poll();

            // If the current number is beyond the integer range, return -1.
            if (currentNumber > Integer.MAX_VALUE) {
                return -1;
            }

            // If the current number is greater than k and is a multiple of k, return it as an integer.
            if (currentNumber > k && currentNumber % k == 0) {
                return (int) currentNumber;
            }

            // Add new candidates to the queue by appending digit1 and digit2 to the current number.
            queue.offer(currentNumber * 10 + digit1);

            // Only add a candidate with digit2 if it is different from digit1.
            if (digit1 != digit2) {
                queue.offer(currentNumber * 10 + digit2);
            }
        }
    }
}
```

C++ Solution

```cpp
#include <queue>

class Solution {
public:
    int findInteger(int k, int digit1, int digit2) {
        // If both digits are 0, return -1 since we cannot form a valid number.
        if (digit1 == 0 && digit2 == 0) {
            return -1;
        }

        // Ensure digit1 is the smaller digit for simplicity.
        if (digit1 > digit2) {
            std::swap(digit1, digit2);
        }

        // Initialize a queue to perform a breadth-first search.
        std::queue<long long> queue;
        queue.push(0);

        while (true) {
            // Retrieve the front element of the queue.
            long long currentNumber = queue.front();
            queue.pop();

            // Check if the current number exceeds the maximum value of int.
            if (currentNumber > INT_MAX) {
                return -1;
            }

            // If the current number is greater than k and is divisible by k, return it.
            if (currentNumber > k && currentNumber % k == 0) {
                return static_cast<int>(currentNumber);
            }

            // Append digit1 to the end of the current number and add it to the queue.
            queue.push(currentNumber * 10 + digit1);

            // If digit1 is not equal to digit2, also append digit2 and add to the queue.
            if (digit1 != digit2) {
                queue.push(currentNumber * 10 + digit2);
            }
        }
    }
};
```

Typescript Solution

```typescript
// Required to use the JavaScript built-in data type representing the largest possible integer.
const MAX_INT: number = Number.MAX_SAFE_INTEGER;

// Represents a node in the BFS with a value and a counter representing the digits used.
interface Node {
  value: number;
  digitsUsed: number;
}

// Function to find the smallest integer that is greater than k and divisible by k containing only the digits digit1 and digit2.
function findInteger(k: number, digit1: number, digit2: number): number {
  // If both digits are 0, it's impossible to form a valid number.
  if (digit1 === 0 && digit2 === 0) {
    return -1;
  }

  // Ensure digit1 is the smaller digit for simplicity.
  if (digit1 > digit2) {
    [digit1, digit2] = [digit2, digit1];
  }

  // Initialize a queue to perform a breadth-first search (BFS).
  let queue: Node[] = [{ value: 0, digitsUsed: 0 }];

  while (queue.length > 0) {
    // Retrieve and remove the first element of the queue.
    let current: Node = queue.shift()!;

    // If the current number exceeds MAX_INT, continue to the next iteration.
    if (current.value > MAX_INT) {
      continue;
    }

    // If the current number is greater than k and is divisible by k, return it.
    if (current.value > k && current.value % k === 0) {
      return current.value;
    }

    // Append digit1 to the end of the current number and add it to the queue if it doesn't exceed MAX_INT.
    if (current.value * 10 + digit1 <= MAX_INT) {
      queue.push({ value: current.value * 10 + digit1, digitsUsed: current.digitsUsed + 1 });
    }

    // If digit1 is not equal to digit2, also append digit2 and add to the queue if it doesn't exceed MAX_INT.
    if (digit1 !== digit2 && current.value * 10 + digit2 <= MAX_INT) {
      queue.push({ value: current.value * 10 + digit2, digitsUsed: current.digitsUsed + 1 });
    }
  }

  // If we exhaust the queue without finding a valid number, return -1.
  return -1;
}
```

Time and Space Complexity

The given Python code uses a breadth-first search (BFS) approach to find the smallest integer greater than `k` that is also divisible by `k`, with the restriction that the integer can only consist of the digits `digit1` and `digit2`. To analyze the time complexity and space complexity, we'll consider that `n` represents the number of digits in the final result.

Time Complexity

The time complexity of this algorithm depends on the number of nodes explored in the BFS until the solution is found. Each node represents a valid number that can be constructed using the given digits.

- For each number `x` constructed, the code generates up to two new numbers, `x * 10 + digit1` and `x * 10 + digit2`, provided `digit1` is not equal to `digit2`.
- The maximum value for a 32-bit signed integer is `2**31 - 1`, which sets the upper limit for possible numbers that could be generated.
- The number `k` is being incremented by a factor of 10 for each new level of the BFS, so the maximum number of levels in the BFS (representing the number of digits in `x`) is `log10(2**31)`.

Assuming that the smallest solution has `n` digits, each level in the BFS can at most have `2**n` nodes to process (every digit can be either `digit1` or `digit2` if they are distinct).

Therefore, a very loose upper bound on the time complexity is $O(2^n)$, where n is the number of digits in the result. However, this does not account for the upper limit of `2**31 - 1`, which means that the effective time complexity is much lower, as it is also bounded by this maximum integer size and it stops once a valid solution is found.

Since this code includes a check to stop early once a solution is found (`x > k and x % k == 0`), the actual time complexity can be significantly optimized depending on the values of `k`, `digit1`, and `digit2`.

Space Complexity

The space complexity is dictated by the maximum size of the deque `q`, which holds all numbers that are generated but have not yet been processed.

- The queue can hold a maximum of `2**n` numbers before a number divisible by `k` is found or the upper limit for `x` is reached.

Thus, the space complexity is also $O(2^n)$, where n is the number of digits in the result. But just like the time complexity, it is bounded by the 32-bit integer limit and optimized by the early stopping condition for valid solutions.