967. Numbers With Same Consecutive Differences

Medium

Problem Description

The given problem requires us to generate all possible integers of a specified length 'n' where the absolute difference between every pair of consecutive digits is 'k'. The key points to consider are:

1. The length of each integer should be exactly 'n' digits long. 2. Consecutive digits in an integer should have a difference of 'k'.

3. The integers should not contain any leading zeros. 4. The result can be returned in any order.

Breadth-First Search Backtracking

- For example, if n = 3 and k = 7, one possible integer could be 181, where the difference between 8 and 1 is 7.

intuition guide for the DFS approach:

Intuition

To solve this problem, a depth-first search (DFS) algorithm can be leveradged. The intuition behind using DFS is to build each valid integer digit by digit and to backtrack when a digit that does not satisfy the constraints is reached. Here is a step-by-step

1. Start with a digit between 1 and 9 (since we cannot have leading zeros), and treat this as the current integer. 2. Perform DFS to add the next digit to the current integer. This digit must be within the range 0-9 and should differ by 'k' from the last digit of the current integer. 3. If 'k' is zero, ensure that you only add the same digit as previously added to prevent duplicates.

- 4. Repeat the process until the length of the current integer is 'n'. 5. Once an integer of length 'n' is built, add it to the result list. 6. Continue exploring other possibilities by backtracking and trying other digits for each position.
- The DFS continues until all possible combinations that meet the conditions have been explored and added to the result list. By using DFS, we both build the integers and validate them at each step of the process.

Solution Approach The implementation uses a Depth-First Search (DFS) method defined as dfs, a recursive approach, which is suitable for building

combinations and is used to explore each possible number according to the constraints. The dfs function receives three

parameters: n, k, and the current temporary integer t.

digit.

temporary integer t.

Data Structures: List for storing the results (ans).

digits, thus satisfying all the constraints of the problem effectively.

Depth-First Search (DFS): Inside the DFS function:

Base Case: For dfs(1, 1, 1), n is not zero, so we do not add t to ans yet.

Here's a step-by-step explanation of the solution approach based on the provided code: Initializing the Answer List: A list named ans is initialized to store the resulting valid integers.

Base Case: If the remaining length for the integer (n) is zero, the current integer (t) satisfies the length requirement and is

Similarly, if last - k is greater or equal to zero and k is not zero (to avoid duplicate digits when k is zero), another

recursive call dfs(n-1, k, t*10 + last - k) is made to explore the possibility of adding last - k as the next

added to the answer list, ans. Calculating the Next Digit: The last digit is obtained by performing t % 10. 0

Depth-First Search (DFS): The dfs function is defined with the following logic:

- **Exploring Next Digits:** 0
- If last + k is less than or equal to 9, a recursive call dfs(n 1, k, t * 10 + last + k) is made to explore the possibility of adding last + k as the next digit.
- This recursive pattern allows the exploration of all valid combinations of digits where the difference between each consecutive pair is k. Initiating DFS for Each Starting Digit: A loop from 1 to 9 is used to start the process since the numbers cannot have leading

zeros. For each starting digit i, the dfs function is called with n - 1 since one digit is already used, k, and i as the current

Returning the Result: After all the DFS calls are completed, return the list ans which contains all the valid integers meeting

the specified conditions. Pattern used: **DFS** (Depth-First Search)

The recursive nature of DFS helps in building numbers digit by digit while maintaining the difference k between consecutive

Example Walkthrough To illustrate the solution approach, let's consider a small example where n = 2 and k = 1. We want to generate all possible integers of length 2 where the absolute difference between every pair of adjacent digits is 1.

Initiating DFS for Each Starting Digit: We iterate through digits 1 to 9 as the first digit of our potential integer because we

∘ Let's choose the starting digit 1. We now call dfs(n - 1, k, t), which translates to dfs(1, 1, 1) since we have used one digit (1) already.

Initializing the Answer List: We start by initializing an empty list ans which will store our results.

cannot have leading zeros.

12).

with this subtraction.

- Calculating the Next Digit: The last digit obtained is 1 (since t % 10 = 1). Exploring Next Digits: ■ Since last + k is 2 (which is less than or equal to 9), we make a recursive call: dfs(n - 1, k, t * 10 + last + k) which is dfs(0, 1,
- Call dfs(1, 1, 2). ■ Possible next digits are 1 (2-1) and 3 (2+1).

For this example, we have iteratively built each possible 2-digit number that follows the absolute difference constraint of 1 by

■ For dfs(0, 1, 12), n is zero, meaning we've built a valid integer of the correct length, hence we add 12 to our answer list ans.

■ Now we also check if we can subtract k from last, but since last - k would be 0 and it's already been considered, we don't proceed

Returning the Result: After all DFS calls with starting digits 1 to 9 are completed, we return the list ans which now contains [12, 21, 23, 32, 34, 43, ..., 89, 98].

exploring each digit starting from 1 to 9 and using the DFS method to build out the integers.

Base case: when the number length becomes 0, we have a complete number

// Main method to find and return all numbers of length n where the difference between

vector<int> result; // Use a more descriptive name for the global answer storage.

// Start from digit 1 to 9, and use these as the starting digits for our numbers.

return result; // Return the global result vector containing all valid numbers.

if (n == 0) { // If the number has reached its target length

// Check if adding 'k' to the last digit remains a digit

result.push_back(current); // Add the number to the result list

int lastDigit = current % 10; // Get the last digit of the current number

// Helper Depth-First Search function to build numbers according to the given constraints.

// Function to find all numbers with unique digits and a difference of 'k' between consecutive digits.

dfs(n - 1, k, start); // Start a DFS for each number of length n with the given 'k'.

dfs(n - 1, k, current * 10 + lastDigit + k); // Form the next number in the sequence

dfs(n-1, k, current * 10 + lastDigit - k); // Form the next number in the sequence

let result: number[] = []; // This now uses camelCase naming and TypeScript syntax for type declaration

// Start from digit 1 to 9, treating these as the initial digits for potential numbers.

// Helper function for Depth-First Search to construct numbers adhering to the constraints.

Start the DFS process from digits 1 to 9 (since the number cannot start with 0)

// Function to find all numbers with unique digits and a fixed difference 'k' between consecutive digits.

// Check if subtracting 'k' from the last digit remains a digit and 'k' is non-zero to avoid repeats

If adding k to the last digit fills the condition (digit stays between 0 to 9)

depth_first_search(num_length - 1, k, current_number * 10 + last_digit + k)

We proceed to the next starting digit 2 and repeat the DFS steps:

Data Structures: List (ans) for storing the results.

Pattern used: **DFS** (Depth-First Search)

Solution Implementation

from typing import List

class Solution:

Python

■ For 1, call dfs(0, 1, 21) and add 21 to ans.

■ For 3, call dfs(0, 1, 23) and add 23 to ans.

Continue this process with all starting digits from 1 to 9.

def numsSameConsecDiff(self, n: int, k: int) -> List[int]:

results.append(current_number)

last_digit = current_number % 10

depth_first_search(n - 1, k, i)

Get the last digit of the current number

Return the list of results after DFS completion

Initialize a list to hold the final results results = [] # Define a depth-first search function for constructing numbers def depth_first_search(num_length: int, k: int, current_number: int):

print(solution.numsSameConsecDiff(3, 7)) # This will print all the numbers of length 3 with consecutive differences of 7.

If subtracting k from the last digit fills the condition and k is not 0 (to prevent duplicates) if last_digit - k >= 0 and k != 0: depth_first_search(num_length - 1, k, current_number * 10 + last_digit - k) # Start the DFS process from digits 1 to 9 (since the number cannot start with 0)

// every two consecutive digits is k

public int[] numsSameConsecDiff(int n, int k) {

vector<int> numsSameConsecDiff(int n, int k) {

return; // Exit the recursion

if (lastDigit - k >= 0 && k != 0) {

for (let start = 1; start <= 9; ++start) {</pre>

function numsSameConsecDiff(n: number, k: number): number[] {

void dfs(int n, int k, int current) {

if (lastDigit + k < 10) {</pre>

for (int start = 1; start <= 9; ++start) {</pre>

List<Integer> results = new ArrayList<>();

for i in range(1, 10):

return results

if last_digit + k <= 9:</pre>

if num_length == 0:

return

```
Java
class Solution {
```

Example usage:

class Solution {

public:

solution = Solution()

```
// Starting from 1 because numbers cannot have leading zeros
        for (int i = 1; i < 10; ++i) {
           depthFirstSearch(n - 1, k, i, results);
       // Convert the list to an array
       int[] answer = new int[results.size()];
        for (int i = 0; i < results.size(); ++i) {</pre>
           answer[i] = results.get(i);
       return answer;
   // Helper function for depth-first search
   private void depthFirstSearch(int remainingDigits, int diff, int currentNum, List<Integer> results) {
       // Base case: if no more digits are needed, add the current number to the result
       if (remainingDigits == 0) {
            results.add(currentNum);
           return;
       // Get the last digit of the current number
        int lastDigit = currentNum % 10;
       // Check if we can add 'diff' to the last digit without exceeding 9
       if (lastDigit + diff <= 9) {</pre>
            depthFirstSearch(remainingDigits - 1, diff, currentNum * 10 + lastDigit + diff, results);
       // Check if we can subtract 'diff' from the last digit without going below 0
       // Also, avoid repeating the same number if diff is 0
       if (lastDigit - diff >= 0 && diff != 0) {
            depthFirstSearch(remainingDigits -1, diff, currentNum *10 + lastDigit - diff, results);
C++
```

return result; // Return the populated result array containing all valid numbers.

};

TypeScript

```
function dfs(n: number, k: number, current: number): void {
      if (n === 0) { // Base case: if the number has reached its required length,
          result.push(current); // it is added to the result array.
          return; // Exit the current recursive call.
      const lastDigit = current % 10; // Extract the last digit of the current number.
      // If the new digit after adding 'k' is still in the range [0, 9],
      if (lastDigit + k < 10) {</pre>
          dfs(n - 1, k, current * 10 + lastDigit + k); // Construct and explore this next number in the sequence.
      // If the new digit after subtracting 'k' is in the range [0, 9] and 'k' is not zero (to avoid duplicates),
      if (lastDigit - k >= 0 && k !== 0) {
          dfs(n - 1, k, current * 10 + lastDigit - k); // Construct and explore this next number in the sequence.
  // Example of calling the method:
  // const uniqueNumbers = numsSameConsecDiff(3, 7);
from typing import List
class Solution:
   def numsSameConsecDiff(self, n: int, k: int) -> List[int]:
       # Initialize a list to hold the final results
        results = []
       # Define a depth-first search function for constructing numbers
       def depth_first_search(num_length: int, k: int, current_number: int):
            # Base case: when the number length becomes 0, we have a complete number
            if num length == 0:
                results.append(current_number)
               return
            # Get the last digit of the current number
            last_digit = current_number % 10
            # If adding k to the last digit fills the condition (digit stays between 0 to 9)
            if last digit + k <= 9:</pre>
                depth_first_search(num_length - 1, k, current_number * 10 + last_digit + k)
            # If subtracting k from the last digit fills the condition and k is not 0 (to prevent duplicates)
            if last_digit - k >= 0 and k != 0:
                depth_first_search(num_length - 1, k, current_number * 10 + last_digit - k)
```

dfs(n - 1, k, start); // Initiate a Depth-First Search for each number of length 'n' matching difference 'k'.

The given Python code defines a function numsSameConsecDiff that generates numbers of length n having a difference of k between every two consecutive digits. The function uses Depth-First Search (DFS) with pruning. For each starting digit from 1 to 9 (i in the loop), the DFS function is called. Inside dfs, the recursive call reduces the n by 1 until it

Time and Space Complexity

for i in range(1, 10):

return results

Example usage:

solution = Solution()

Time Complexity

depth_first_search(n - 1, k, i)

Return the list of results after DFS completion

reaches 0, at which point a number is formed and added to the answer list (ans). There are two recursive calls in the function, one for last + k and another for last - k, however, when k is 0, they result in the same digit being added, so only one recursive call is need in that case, effectively pruning the search space.

print(solution.numsSameConsecDiff(3, 7)) # This will print all the numbers of length 3 with consecutive differences of 7.

The worst-case time complexity is when k is not 0 and we have to explore both increasing and decreasing options at each step, which gives us 0(2ⁿ) - this is because, at each step, you have a possibility of two different digits to be appended, effectively doubling the possibilities with each additional digit until reaching the desired length n. However, the actual time complexity might be less than 0(2ⁿ) due to two factors:

2. There's no branching when k = 0, since last + k and last - k result in the same digit. **Space Complexity**

1. Pruning occurs when last + k > 9 or last - k < 0, in which case the respective recursive call is not made.

The space complexity of the code is O(n). This is because of the call stack that is used when the dfs function is recursively called. The depth of the recursive call stack will at most be n, where n is the length of the number we want to create. The list ans

also uses space, but it doesn't contribute more than 0(10^n) in space complexity, since we're only generating number sequences of length n starting with a non-zero digit, which in the worst-case gives us all permutations of n-length numbers starting with non-zero digits. However, this is a constant factor when considering space complexity with respect to n.

Thus, the overall space complexity is 0(n) from the recursive call stack. Meanwhile, the space to hold the generated numbers is dependent on the number of valid combinations found, which is not directly dependent on n, but rather a combination of both n and k.