2844. Minimum Operations to Make a Special Number

```
Greedy Math
Medium
                        String
                               Enumeration
```

**Problem Description** 

The problem requires us to transform a given string num that represents a non-negative integer into a special number using a series of operations. A number is deemed special if it's divisible by 25. Every operation involves choosing and deleting any single digit from num, and this process can be done as many times as needed. If we end up deleting all digits, the string num becomes '0', which is a valid special number since 0 is divisible by 25. The ultimate goal is to find the minimum number of operations that make num a special number.

#### To find the minimum number of operations needed, we need to think about the properties of numbers divisible by 25. Any number divisible by 25 ends with '00', '25', '50', or '75'. Therefore, our task is to rearrange the digits of num by deleting some of them to

Intuition

get a number ending with these two digits. Let's focus on the two least significant digits in our operations because any digit(s) before them won't affect divisibility by 25 (e.g., 125, 225, 1025 - all end with 25 and thus are special). For num to be special, it must end with any of the pairs mentioned.

With this in mind, we look for the occurrence of these pairs within num, and aim to remove as few digits as possible to form a special number. We can achieve this using a depth-first search (DFS) strategy. We explore each position i in the string num, where we either: 1. Skip the current digit (which is equivalent to removing it) and recursively continue with the next digit.

Using this approach, we can recursively calculate the minimum operations needed and employ memoization to avoid

recalculations (utilize previously solved sub-problems). Memoization is applied through the @cache decorator on the DFS

2. Take the current digit into consideration, add it to our forming number and check the number's remainder when divided by 25.

function, which stores results of function calls with particular arguments for fast retrieval. The recursion continues until we've checked every digit in <a href="num">num</a> (i == n), at which point, if we've formed a number divisible by 25

minimum of these recursive calls at each step gives the least operations needed.

The core function dfs(i: int, k: int) -> int plays a crucial role in this approach:

Solution Approach

(k == 0), we need 0 more operations, otherwise, we haven't formed a special number and return a high operation count (n). The

The solution makes use of a recursive depth-first search (DFS) combined with memoization to optimize the computation. DFS is a powerful algorithmic technique for exploring all the options in a decision tree-like structure, one path at a time, until a goal state is reached.

#### • i tracks the current digit's index within num. • k is the current number formed by the selected digits from num, modulo 25. We use modulo 25 because we only care if the number formed is

pairs.

Within dfs, we consider two options at each digit:

because skipping the digit is an operation.

divisible by 25, not the actual number. The base case for the recursion occurs when i equals the length of num (n). If k is 0 at this point, we've successfully formed a special number, so no more operations are needed, and 0 is returned. Otherwise, we return n, which is a large number signifying

an unsuccessful attempt to form a special number (since we cannot perform more operations than the number of digits).

2. Include the current digit: We call dfs(i + 1, (k \* 10 + int(num[i])) % 25) to check if adding the current digit (int(num[i])) brings us closer to making the number special. We perform modulo 25 on the result to update k. Then, we use the min function to choose the option with the fewer operations. Memoization is applied using the @cache decorator from Python's standard library, which effectively caches the results of the dfs function calls with specific (i, k)

1. Skip the current digit: We call dfs(i + 1, k) to continue to the next index while keeping the current value of k. We also add 1 to the result

The algorithm initiates with dfs(0, 0), meaning we start with the first digit and a k value of 0, indicating no number formed yet. Finally, the outer dfs call returns the minimum number of operations required to achieve the task. With this solution, we examine each possibility while ensuring we do not re-compute the same state (combination of i and k)

The data structure used here is implicit within the function call stack for the recursive approach, and the caching mechanism uses some form of hash table to store the results.

multiple times, thereby significantly reducing the time complexity compared to a plain recursive approach without memoization.

end this number with '00', '25', '50', or '75'. We start with the recursive depth-first search (DFS) function dfs(i, k). Initially, we call dfs(0, 0) with i = 0 (which is the first index of the string) and k = 0 (no number formed yet).

The goal is to minimize the operations to make "3527" a special number (divisible by 25). To do so, we need to find a way to

For the first digit '3', the initial call is dfs(0, 0). Two recursive paths will be explored: dfs(1, 0) (skipping '3') and dfs(1, 3

Let's assume we first explore including the digit '3'. If we then include '5', the next steps are  $dfs(2, (3 * 10 + 5) % 25) \Rightarrow$ 

 $dfs(2, 35 \% 25) \Rightarrow dfs(2, 10)$ . It looks like we won't end with '00', '25', '50', or '75' if we keep going this route. So we may

By applying the same decision logic to all possibilities, we find that to end with '25', we can skip '3' and keep '527'. The

d. dfs(4, (2 \* 10 + 7) % 25) by including '7'  $\Rightarrow$  this is dfs(4, 0). At i = 4, which is the length of num, we check if k is 0,

which it is. Therefore, we've formed the number '527', which is divisible by 25, so no more operations are needed, and the

If we had instead tried other combinations to end with '00', '50', or '75', we would have found that none of these paths gives a

### % 25) (including '3').

special number.

**Python** 

class Solution:

Solution Implementation

from functools import lru\_cache

@lru cache(maxsize=None)

if index == n:

def dfs(index: int, remainder: int) -> int:

return 0 if remainder == 0 else n

exclude\_current = dfs(index + 1, remainder) + 1

best\_option = min(exclude\_current, include\_current)

# Base case: if we reached the end of the number string

# Compute new remainder when the current digit is included

# Start DFS from the first digit, with remainder initialized to 0

std::function<int(int, int)> dfs = [&](int i, int k) -> int {

// Depth-First Search function to compute the minimum operations

const dfs = (currentIndex: number, remainder: number): number => {

// Check if we already calculated the result for this subproblem

// Case 1: Skip the current digit, which costs us one operation

// compare with the result from skipping and choose the minimum

memo[currentIndex][remainder] = dfs(currentIndex + 1, remainder) + 1;

// Case 2: Take the current digit and update the remainder accordingly,

dfs(currentIndex + 1, (remainder \* 10 + Number(num[currentIndex])) % 25)

memo[i][k] = dfs(i + 1, k) + 1;

function minimumOperations(num: string): number {

if (currentIndex === length) {

return remainder === 0 ? 0 : length;

if (memo[currentIndex][remainder] !== -1) {

memo[currentIndex][remainder] = Math.min(

// Return the computed minimum for this subproblem

// Start the DFS from the first index with a remainder of 0

memo[currentIndex][remainder],

return memo[currentIndex][remainder];

return memo[currentIndex][remainder];

const length = num.length;

if (i == n) { // Base case: If we have considered all digits

return memo[i][k]; // Return the calculated value

if  $(memo[i][k] != -1) { // If we have already calculated this state}$ 

memo[i][k] = std::min(memo[i][k], dfs(i + 1, (k \* 10 + num[i] - '0') % 25));

// If we skip the current digit, increment the operations count

return memo[i][k]; // Return the minimum operations for this state

return dfs(0, 0); // Start the DFS from the first digit with a remainder of 0

// Create a memoization table 'memo', initialized with -1, to store results of subproblems

// Otherwise, the string cannot be made divisible by 25 using these digits

const memo: number[][] = Array.from({ length: length }, () => Array.from({ length: 25 }, () => -1));

// If we are at the end of the string and remainder is 0, no more operations are needed

# If current remainder is 0, it means we can form a number divisible by 25,

# Case 1: Exclude the current digit, increment the operations counter

# Case 2: Include the current digit, no increment to operations counter

include\_current = dfs(index + 1, (remainder \* 10 + int(num[index])) % 25)

# Choose the minimum operations between including or excluding the current digit

# thus no more operations needed. Otherwise, we can't form such number hence return n

**Example Walkthrough** 

backtrack and try skipping some digits instead.

c. dfs(3, (5 \* 10 + 2) % 25) by including '2'  $\Rightarrow$  this is dfs(3, 2).

a. Skip the current digit: dfs(i + 1, k) and add 1 to the operations count.

b. Include the current digit: dfs(i + 1, (k \* 10 + int(num[i])) % 25).

At each step of the recursion, we have two choices:

Let's consider an example where the input string num is "3527". Following the solution approach:

- operations would look like this: a. dfs(1, 0) by skipping '3' (operations = 1). b. dfs(2, (0 \* 10 + 5) % 25) by including '5' (operations still = 1)  $\Rightarrow$  this is dfs(2, 5).
- lower minimum operation count than ending with '25'. The minimal number of operations is memorized for each state (i, k) to avoid redundant calculations. Therefore, the answer for this particular num "3527" is 1. We perform one deletion operation (removing '3'), and then '527' is a

minimum number of operations is the one obtained from this branch of the decision tree, which is 1.

def minimum operations(self, num: str) -> int: # Calculate the length of the input number string n = len(num)# Define a Depth First Search with memoization using the lru\_cache decorator

#### # Example use case solution = Solution() print(solution.minimum\_operations("123")) # Should output the minimum operations to get a multiple of 25

private String number;

private int length;

Java

class Solution {

return dfs(0, 0)

return best\_option

private Integer[][] memoization;

```
// This method initializes the problem and calls the depth-first search method to find the solution.
    public int minimumOperations(String num) {
        length = num.length();
        this.number = num;
        memoization = new Integer[length][25]; // 25 here because any two digits modulo 25 cover all possible remainders.
        return depthFirstSearch(0, 0);
    // This is a recursive depth-first search method which tries to find the minimum operations to get a multiple of 25.
    private int depthFirstSearch(int index, int remainder) {
       // Base case: if we've reached the end of the string
        if (index == length) {
            // We return 0 if remainder is 0, meaning the current sequence is a multiple of 25. Otherwise, we return a high cost.
            return remainder == 0 ? 0 : length;
        // Memoization to avoid recalculating subproblems
        if (memoization[index][remainder] != null) {
            return memoization[index][remainder];
        // Option 1: Do not include the current index in our number and move to the next digit
        memoization[index][remainder] = depthFirstSearch(index + 1, remainder) + 1;
        // Option 2: Include the current index in the number, update the remainder, and see if it leads to a better solution
        int nextRemainder = (remainder * 10 + number.charAt(index) - '0') % 25;
        memoization[index][remainder] = Math.min(memoization[index][remainder], depthFirstSearch(index + 1, nextRemainder));
        // Return the computed minimum operations for the current subproblem
        return memoization[index][remainder];
C++
#include <vector>
#include <string>
#include <cstring> // For memset
#include <functional> // For std::function
class Solution {
public:
    int minimumOperations(std::string num) {
        int n = num.size(); // Get the size of the input string
        std::vector<std::vector<int>> memo(n, std::vector<int>(25, -1)); // Create a memoization table with -1 as default values
```

// Define a recursive lambda function to find the minimum operations. It takes the current position and a remainder 'k'

// Try including the current digit and update the remainder, choose the minimum between skipping and taking this digit

return k == 0 ? 0 : n; // If remainder is 0, return 0 operations, else return n (max operations)

**}**;

return dfs(0, 0);

**}**;

**TypeScript** 

**}**;

```
from functools import lru_cache
class Solution:
    def minimum operations(self, num: str) -> int:
        # Calculate the length of the input number string
        n = len(num)
        # Define a Depth First Search with memoization using the lru_cache decorator
        @lru cache(maxsize=None)
        def dfs(index: int, remainder: int) -> int:
           # Base case: if we reached the end of the number string
           if index == n:
               # If current remainder is 0, it means we can form a number divisible by 25,
                # thus no more operations needed. Otherwise, we can't form such number hence return n
                return 0 if remainder == 0 else n
           # Case 1: Exclude the current digit, increment the operations counter
            exclude current = dfs(index + 1, remainder) + 1
           # Case 2: Include the current digit, no increment to operations counter
           # Compute new remainder when the current digit is included
            include_current = dfs(index + 1, (remainder * 10 + int(num[index])) % 25)
            # Choose the minimum operations between including or excluding the current digit
            best_option = min(exclude_current, include_current)
            return best_option
        # Start DFS from the first digit, with remainder initialized to 0
        return dfs(0, 0)
# Example use case
solution = Solution()
print(solution.minimum_operations("123")) # Should output the minimum operations to get a multiple of 25
Time and Space Complexity
```

## The given code defines a recursive function dfs with memoization to solve the problem. Let's analyze the time complexity:

# • There are n digits in the input string num, so there are n levels of recursion.

**Time Complexity** 

• At each level i, there are two choices: either take the current digit into consideration by calculating (k \* 10 + int(num[i])) % 25 or skip it. • Memoization is used to avoid recalculating the states by storing results for each unique (i, k). The parameter i can take n different values (from 0 to n - 1) and k can take at most 25 different values (from 0 to 24) since we are only interested in the remainder when divided by 25.

- Due to memoization, each state is computed only once. Therefore, the time complexity is 0(n \* 25), which simplifies to 0(n)because the 25 is a constant factor.
- **Space Complexity**
- Due to memoization, we store results for each unique (i, k). Since i can take n different values and k can take 25 different values, the space required for memoization is 0(n \* 25), which simplifies to 0(n). • The recursion depth can go up to n levels, because we make a decision for each digit in the string num.

The space complexity is determined by the storage required for memoization and the depth of the recursion stack:

Taking these into consideration, the overall space complexity of the algorithm is 0(n) due to memoization and the recursion stack.