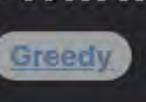
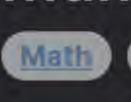
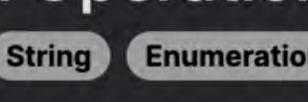
# 2844. Minimum Operations to Make a Special Number

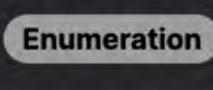












Leetcode Link

# **Problem Description**

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.

The problem requires us to transform a given string num that represents a non-negative integer into a special number using a series

### To find the minimum number of operations needed, we need to think about the properties of numbers divisible by 25. Any number

Intuition

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

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 get a

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.

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.

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

- Using this approach, we can recursively calculate the minimum operations needed and employ memoization to avoid recalculations
- results of function calls with particular arguments for fast retrieval.

The recursion continues until we've checked every digit in num (i == n), at which point, if we've formed a number divisible by 25 (k ==

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

(utilize previously solved sub-problems). Memoization is applied through the @cache decorator on the DFS function, which stores

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

Solution Approach 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

#### The core function $dfs(i:int, k:int) \rightarrow int$ plays a crucial role in this approach:

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 divisible by 25, not the actual number.

number, so no more operations are needed, and 0 is returned. Otherwise, we return n, which is a large number signifying an

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

Within dfs, we consider two options at each digit:

brings us closer to making the number special. We perform modulo 25 on the result to update k.

Finally, the outer dfs call returns the minimum number of operations required to achieve the task.

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

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

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

the result because skipping the digit is an operation.

some form of hash table to store the results.

index of the string) and k = 0 (no number formed yet).

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

backtrack and try skipping some digits instead.

a. dfs(1, 0) by skipping '3' (operations = 1).

operations would look like this:

from functools import lru\_cache

n = len(num)

@lru\_cache(maxsize=None)

if index == n:

return best\_option

return dfs(0, 0)

def minimum\_operations(self, num: str) -> int:

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

# Calculate the length of the input number string

return 0 if remainder == 0 else n

class Solution:

from Python's standard library, which effectively caches the results of the dfs function calls with specific (i, k) pairs. 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.

Then, we use the min function to choose the option with the fewer operations. Memoization is applied using the @cache decorator

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]))

With this solution, we examine each possibility while ensuring we do not re-compute the same state (combination of i and k) multiple times, thereby significantly reducing the time complexity compared to a plain recursive approach without memoization.

The data structure used here is implicit within the function call stack for the recursive approach, and the caching mechanism uses

**Example Walkthrough** 

this number with '00', '25', '50', or '75'. 2. 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

1. 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 end

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

25) (including '3').

4. 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 %

b. dfs(2, (0 \* 10 + 5) % 25) by including '5' (operations still = 1)  $\Rightarrow$  this is dfs(2, 5).

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

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

- 5. 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) →  $dfs(2, 35 \% 25) \Rightarrow dfs(2, 10)$ . It looks like we won't end with '00', '25', '50', or '75' if we keep going this route. So we may
- c. dfs(3, (5 \* 10 + 2) % 25) by including '2'  $\Rightarrow$  this is dfs(3, 2). 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 minimum

7. 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

Therefore, the answer for this particular num "3527" is 1. We perform one deletion operation (removing '3'), and then '527' is a special

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

- lower minimum operation count than ending with '25'. 8. The minimal number of operations is memorized for each state (i, k) to avoid redundant calculations.
- **Python Solution**
- # Case 1: Exclude the current digit, increment the operations counter 17 exclude\_current = dfs(index + 1, remainder) + 1 18 19 20 # Case 2: Include the current digit, no increment to operations counter

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

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

# 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

# Define a Depth First Search with memoization using the lru\_cache decorator

# 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

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

```
32 # Example use case
33 solution = Solution()
   print(solution.minimum_operations("123")) # Should output the minimum operations to get a multiple of 25
35
```

number.

11

12

13

23

24

25

26

27

28

29

30

31

29

30

31

32

33

35

10

11

12

13

14

15

16

18

19

20

21

22

23

24

26

31 }

32

34 }

```
Java Solution
1 class Solution {
       private Integer[][] memoization;
       private String number;
       private int length;
6
       // 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.
10
11
           return depthFirstSearch(0, 0);
12
13
       // 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
16
           if (index == length) {
17
               // We return 0 if remainder is 0, meaning the current sequence is a multiple of 25. Otherwise, we return a high cost.
18
19
               return remainder == 0 ? 0 : length;
20
           // Memoization to avoid recalculating subproblems
22
           if (memoization[index][remainder] != null) {
23
               return memoization[index][remainder];
24
25
           // Option 1: Do not include the current index in our number and move to the next digit
26
           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
27
           int nextRemainder = (remainder * 10 + number.charAt(index) - '0') % 25;
28
```

// Return the computed minimum operations for the current subproblem

return memoization[index][remainder];

#### 10 11 12 13

C++ Solution

1 #include <vector>

2 #include <string>

6 class Solution {

#include <cstring> // For memset

#include <functional> // For std::function

return remainder === 0 ? 0 : length;

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

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

// Return the computed minimum for this subproblem

memo[currentIndex][remainder],

return memo[currentIndex][remainder];

return memo[currentIndex][remainder];

// 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)

```
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'
            std::function<int(int, int)> dfs = [&](int i, int k) -> int {
                 if (i == n) { // Base case: If we have considered all digits
 14
                     return k == 0 ? 0 : n; // If remainder is 0, return 0 operations, else return n (max operations)
 15
 16
 17
                 if (memo[i][k] != -1) { // If we have already calculated this state
                    return memo[i][k]; // Return the calculated value
 18
 19
                // If we skip the current digit, increment the operations count
 20
                 memo[i][k] = dfs(i + 1, k) + 1;
 21
 22
                 // Try including the current digit and update the remainder, choose the minimum between skipping and taking this digit
                 memo[i][k] = std::min(memo[i][k], dfs(i + 1, (k * 10 + num[i] - '0') % 25));
 23
 24
                 return memo[i][k]; // Return the minimum operations for this state
            };
 25
 26
 27
             return dfs(0, 0); // Start the DFS from the first digit with a remainder of 0
 28
 29 };
 30
Typescript Solution
   function minimumOperations(num: string): number {
       const length = num.length;
       // Create a memoization table 'memo', initialized with -1, to store results of subproblems
       const memo: number[][] = Array.from({ length: length }, () => Array.from({ length: 25 }, () => -1));
       // Depth-First Search function to compute the minimum operations
       const dfs = (currentIndex: number, remainder: number): number => {
           // If we are at the end of the string and remainder is 0, no more operations are needed
           // Otherwise, the string cannot be made divisible by 25 using these digits
           if (currentIndex === length) {
```

memoization[index][remainder] = Math.min(memoization[index][remainder], depthFirstSearch(index + 1, nextRemainder));

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

25 or skip it.

Space Complexity

Time and Space Complexity

);

27 }; 28 // Start the DFS from the first index with a remainder of 0 29 30 return dfs(0, 0);

At each level i, there are two choices: either take the current digit into consideration by calculating (k \* 10 + int(num[i])) %

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

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.

Memoization is used to avoid recalculating the states by storing results for each unique (i, k). The parameter i can take n

because the 25 is a constant factor.

Due to memoization, each state is computed only once. Therefore, the time complexity is 0(n \* 25), which simplifies to 0(n)

- The space complexity is determined by the storage required for memoization and the depth of the recursion stack: • Due to memoization, we store results for each unique (i, k). Since i can take n different values and k can take 25 different
- The recursion depth can go up to n levels, because we make a decision for each digit in the string num.

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

values, the space required for memoization is 0(n \* 25), which simplifies to 0(n).