753. Cracking the Safe

Problem Description

password within the most recent n digits and unlock.

Depth-First Search Graph

Eulerian Circuit

In this problem, we're tasked with finding a string that will eventually unlock a safe that is protected by a password. The password itself is a sequence of n digits, and each digit ranges from 0 to k - 1. The safe uses a sliding window to check the correctness of the entered password, comparing the most recent n digits entered against the actual password.

"123", "234", and finally "345". Only the last sequence "345" matches the password, so the safe unlocks. The goal is to generate the shortest possible string such that, as we type in this string, the safe will eventually identify the correct

For instance, if the password of the safe is "345" (n=3), and you enter the sequence "012345", the safe checks "0", "01", "012",

Intuition

The solution to this problem uses a depth-first search (DFS) algorithm to build a sequence that contains every possible

Hard

represents a (n-1)-digit sequence, and each directed edge represents a possible next digit that can be added to transform one (n-1)-digit sequence into a n-digit sequence. The intuition is to visit each possible combination (every 'node') exactly once. Starting with the sequence "0" repeated n-1 times, the algorithm adds one digit at a time by traversing unvisited edges (using the remaining k possible digits), and then moves onto

combination of n digits with digits from 0 to k - 1. This is akin to finding an Eulerian path or circuit in a graph, where each node

A hash set (vis) is used to keep track of visited combinations, ensuring that each possible n-digit combination is entered exactly once. The mod operation (e % mod) in the DFS function is used to obtain the next (n-1)-digit sequence (the 'node') after adding a

the next sequence by removing the first digit and appending a new one, much like in a sliding window approach.

new digit. The DFS continues until all possible sequences are visited. At the end, the function returns the constructed string which will include the necessary sequence to unlock the safe at some point when typed in.

The solution uses a Depth-First Search (DFS) algorithm to explore the sequences of digits that will unlock the safe. Here's an explanation of the implementation:

Depth-First Search (DFS) Function: The core of the solution is the dfs function, which attempts to visit all n-digit sequences

exactly once. Each call to dfs handles a particular (n-1)-digit sequence (u), trying to append each of the k possible next digits (x) to form a n-digit sequence.

Solution Approach

Handling Combinations: Each time we consider adding a digit x to sequence u, we form a new sequence (e) by doing u * 10 + x. This operation essentially shifts the digits left and appends the new digit to the right.

- Visited Sequences Tracking: A set named vis is used to keep track of visited n-digit sequences. If a sequence e has not been visited yet (e not in vis), it's added to the vis set to mark it as visited. Sliding to Next Sequence: After marking e as visited, we calculate the next (n-1)-digit sequence v to continue the DFS. This
- is done via e % mod, where mod = 10 ** (n 1). The modulo operation essentially removes the leftmost digit from e (because we've already handled this n-digit sequence).

Building the Result: As we explore each digit x that can be added to u, we append x to ans, which is a list that will eventually

Finalizing the Result: Once the DFS is complete, we append "0" repeated n-1 times to ans to represent the initial state where

the safe started checking the digits. We leave the DFS function with DFS(0), which inputs the first valid n-1 digit combination.

Initialization and Starting the DFS: We initialize ans and vis, and then we start the DFS from the initial sequence "0" repeated n-1 times (dfs(0)). This assumes that the preceding sequence of digits ends with zeros, which is a neutral assumption since we seek any valid sequence to unlock the safe.

contain all digits of the minimum length string necessary to unlock the safe.

Concatenating the Digits: Finally, we join all the digits in ans together into a string ("".join(ans)) and return this string as the solution.

This approach guarantees that we generate a string with all possible n-digit combinations that can occur as the safe checks the

last n digits entered. It ensures that somewhere in this string is the correct sequence to unlock the safe, effectively solving the

problem. **Example Walkthrough**

digit may range from 0 to k-1 where k=2. This means each digit in the password can either be 0 or 1.

DFS from "0": During the DFS, we try appending each digit.

• "10" is not visited, hence we mark it as visited and add it to ans.

• "11" is not in vis. We mark it as visited and it's traced in ans.

zero to the beginning: 0 + ans = [0, 0, 1, 0, 1].

def crackSafe(self, n: int, k: int) -> str:

Calculate the modulus for finding vertices

Initialize a set to keep track of visited edges

// We use a HashSet to keep track of visited nodes during DFS

// The modulo for trimming the prefix when moving to the next node

* Starts the process to find the sequence that cracks the safe.

st @param k the range of digits from 0 to k-1 that can be used in the combination

* @return a string representing the minimum length sequence that cracks the safe

// Calculate the modulus to use for creating n-1 length prefixes

* @param n the number of digits in the safe's combination

// Append the initial prefix to complete the sequence

public String crackSafe(int n, int k) {

// Start DFS from node 0

modulus = (int) Math.pow(10, n - 1);

for (int i = 0; i < n - 1; i++) {

answer.append("0");

return answer.toString();

def dfs(current_vertex):

modulus = 10 ** (n - 1)

return "".join(answer)

for x in range(k):

Now let's walk through how the DFS algorithm would find the shortest possible string that includes the password:

Let's illustrate the solution approach using an example. Suppose the password of the safe is a sequence of n = 2 digits, and each

Initialization: We initialize ans as an empty list to store our result, and vis as an empty set to track visited sequences. We're aiming to start DFS with an initial sequence of n-1 digit, which is "0". Starting with "0": We start with the sequence "0", and we will try to append 0 and 1 to it in the DFS.

First, we append 0 to "0" making it "00". Since "00" hasn't been visited, we mark it as visited and then the DFS considers "0" and tries to append 1.

We try appending 0 to "1", yielding "10".

Now we try appending 1 to "1", resulting in "11".

 Next, we append 1 to "0" which gives us "01". Again, "01" hasn't been visited, so we mark it and consider the next sequence which only contains "1". **DFS from "1"**: Now our sequence is "1", and we repeat the process.

Final Sequence: To complete the string, we need to consider an initial state represented by n-1 zeros. Since n=2, we add one

Tracking visited combinations: At this point, all possible n-digit combinations have been visited: "00", "01", "10", "11". Building the Result: Our ans list consists of the digits we added to sequences in the DFS order: [0, 1, 0, 1].

Concatenating into a String: We join all the digits in ans to form the string "00101".

within this string as the safe's sliding window checks the most recent n digits entered.

- This string "00101" is the shortest sequence that ensures the safe will unlock, as it includes all possible combinations of the n digits "00", "01", "10", "11" within its consecutive characters. If the actual password were "00", "01", "10", or "11", it would be found
- **Python** class Solution:
- visited.add(edge) # Mark the edge as visited next_vertex = edge % modulus # Calculate the next vertex dfs(next_vertex) # Recursive call to visit the next vertex answer.append(str(x)) # Append the current character to the answer

Depth-first search function to traverse through the nodes/vertices

edge = current_vertex * 10 + x # Forming a new edge

if edge not in visited: # If the edge is not visited

visited = set() # List to store the characters of the answer answer = []dfs(0) # Start DFS from the vertex 0 # Append the initial combination to the answer (n-1 zeros) answer_append("0" * (n - 1)) # Join all characters to form the final answer string and return

```
private Set<Integer> visited = new HashSet<>();
// StringBuilder to store the answer
private StringBuilder answer = new StringBuilder();
```

private int modulus;

dfs(0, k);

class Solution {

/**

*/

Java

Solution Implementation

```
* Performs DFS to find the Eulerian path/circuit in the De Bruijn graph.
    * @param node the current node in the graph
    * @param k the range of digits from 0 to k-1
    */
    private void dfs(int node, int k) {
       // Try all possible next digits to create an edge from the current node
        for (int x = 0; x < k; ++x) {
           // Create the new edge by appending digit x to the current node
            int edge = node * 10 + x;
            // If the edge has not been visited, add it to the visited set
            if (visited.add(edge)) {
                // Calculate the next node by removing the oldest digit (modulus)
                int nextNode = edge % modulus;
                // Perform DFS on the next node
                dfs(nextNode, k);
                // Append the current digit to the answer
                answer.append(x);
C++
#include <functional>
#include <unordered_set>
#include <string>
#include <cmath>
class Solution {
public:
   // Function to generate and return the De Bruijn sequence.
   // 'n' represents the length of the subsequences and
    // 'k' represents the range of digits (0 to k-1).
    string crackSafe(int n, int k) {
       // Create a set to keep track of visited combinations.
       std::unordered_set<int> visited;
       // Compute 10^{(n-1)}, used later to find the next state.
       int mod = std::pow(10, n - 1);
       // Initialize the answer string.
       string result;
        // Depth-first search function to explore all possible combinations.
       // 'u' represents the current combination being explored as a prefix.
       std::function<void(int)> dfs = [&](int u) {
           // Try to append each possible digit from 0 to k-1.
            for (int x = 0; x < k; ++x) {
                // Create the new mixed radix combination by appending the digit x.
                int combo = u * 10 + x;
                // If the combination is not yet visited, continue the exploration.
                if (visited.count(combo) == 0) {
                    // Mark the new combination as visited.
                    visited.insert(combo);
                    // Recursively explore the next state by taking the last (n-1) digits.
                    dfs(combo % mod);
                    // Append the current digit to the result.
                    result.push_back(x + '0');
       };
       // Start the DFS from the combination of n zeroes.
       dfs(0);
       // To close the De Bruijn sequence, append n-1 zeroes at the end.
        result += std::string(n - 1, '0');
       return result;
};
```

class Solution:

dfs(0);

return result;

TypeScript

// Importing necessary utilities from external libraries

type DFSFunction = (u: number) => void;

// is not needed in TypeScript for the described functionality.

// Define the type for our Depth-first search (DFS) function

// Function to generate and return the De Bruijn sequence.

function crackSafe(n: number, k: number): string {

let visited: Set<number> = new Set();

let mod: number = Math.pow(10, n - 1);

let dfs: DFSFunction = (u: number) => {

let combo: number = u * 10 + x;

result += x.toString();

// Start the DFS from the combination of n zeroes.

if (!visited.has(combo)) {

visited.add(combo);

dfs(combo % mod);

result += '0'.repeat(n - 1);

def dfs(current_vertex):

modulus = 10 ** (n - 1)

visited = set()

for x in range(k):

def crackSafe(self, n: int, k: int) -> str:

Calculate the modulus for finding vertices

List to store the characters of the answer

Initialize a set to keep track of visited edges

for (let x = 0; x < k; ++x) {

// Initialize the answer string.

let result: string = "";

// Set to keep track of visited combinations.

// Parameter 'n' represents the length of the subsequences,

// and parameter 'k' represents the range of digits (0 to k-1).

// Compute $10^{(n-1)}$, used later to find the next state.

// Try to append each possible digit from 0 to k-1.

// Mark the new combination as visited.

// Append the current digit to the result.

// To close the De Bruijn sequence, append n-1 zeroes at the end.

// The crackSafe function can now be used globally in any TypeScript file

Depth-first search function to traverse through the nodes/vertices

edge = current_vertex * 10 + x # Forming a new edge

if edge not in visited: # If the edge is not visited

visited.add(edge) # Mark the edge as visited

next_vertex = edge % modulus # Calculate the next vertex

dfs(next_vertex) # Recursive call to visit the next vertex

answer.append(str(x)) # Append the current character to the answer

// Depth-first search function to explore all possible combinations.

// Create the new mixed radix combination by appending the digit x.

// If the combination is not yet visited, continue the exploration.

// Recursively explore the next state by taking the last (n-1) digits.

answer = []dfs(0) # Start DFS from the vertex 0 # Append the initial combination to the answer (n-1 zeros) answer_append("0" * (n - 1)) # Join all characters to form the final answer string and return return "".join(answer) Time and Space Complexity The given Python code aims to generate the shortest string that contains all possible combinations of a given length n using

(DFS) to construct the Eulerian circuit/path in a De Bruijn graph.

Time Complexity The time complexity can be analyzed based on the total number of nodes and edges visited in the DFS. Each node represents a

directed graph with k possible outgoing edges from each node, there will be a total of k^n edges. Therefore, the time complexity of the DFS is O(k^n), as this is the number of edges, and each edge is visited once.

digits from 0 to k-1, which is essentially the De Bruijn sequence problem. In essence, the algorithm performs a Depth-First Search

unique combination of n-1 digits, resulting in $k^{(n-1)}$ possible nodes. The DFS visits each edge exactly once. Since the graph is a

The space complexity is primarily determined by the storage of the visited edges, which is managed by the vis set. Since each

unique combination of n digits is stored as an edge, up to k^n edges can be in the set, leading to a space complexity of O(k^n).

Space Complexity

The function also uses a recursive call stack for DFS, which in the worst case could grow up to the number of edges, i.e., O(k^n) in space.

The ans list stores all the visited edges' last digits plus the padding at the end, accounting for at most k^n + (n-1) elements. However, since k^n dominates n-1 as k^n can grow much larger with increasing n and k, the contribution of n-1 can be considered negligible for large enough k and n.

Overall, the space complexity is O(k^n), dominated by the storage requirements of the vis set and the recursion call stack.