351. Android Unlock Patterns

Medium Dynamic Programming Backtracking Leetcode Link

Problem Description

The problem belongs to the category of backtracking problems that simulate the process of constructing the unlock patterns of an Android device, which has a distinctive `3 x 3` grid with dots. A user has to draw a pattern by connecting dots without repeating any dot. The challenge is to count all unique and valid unlock patterns that consist of at least `m` and at most `n` dots.

The constraints for a pattern to be valid are:

1. **Distinct Dots**: Every dot can only appear once in a pattern.
2. **Line Segment Passing**: If a line connecting two dots passes through the center of another dot, then that center dot must have already been used in the pattern.

The task involves finding all such unique patterns that obey the above rules and counting them, which may sound simple but can be quite complex to approach due to the possibilities and constraints involved.

Intuition

The core idea of the solution is to use backtracking, a common technique to explore all the possibilities in a systematic way. Backtracking allows us to build a solution step-by-step and backtrack as soon as we realize that the current path will not lead to a solution. This way, we efficiently explore the search space of all possible patterns.

To simplify the problem, we use the following strategies:

1. **Symmetry**: The `3 x 3` grid has symmetrical properties that we can exploit to reduce computation. For example, starting from corner points (1, 3, 7, 9) or starting from edge midpoints (2, 4, 6, 8) give rise to patterns that are essentially similar. Therefore, computing for one corner and multiplying the result by 4 (since there are 4 corners) and doing the same for the edge midpoints is sufficient.

2. **Cross Points**: To address the constraint regarding the line segment passing, we create an auxiliary 'cross' matrix that records the cross point for each pair of dots. For instance, the pair (1, 3) has the cross point 2. During the exploration of patterns, we use this information to check if we are allowed to connect two dots straightaway or if we need to have passed through certain dots first.

Approaching the problem this way breaks down the complex task into smaller steps and uses reasoning to efficiently search for all possible solutions by considering the symmetry and constraints unique to the unlock pattern challenge.

In essence, we consider each dot as a starting point and explore all possible patterns emanating from it while keeping track of the visited dots and ensuring we adhere to the rules stated. We recursively build paths and count those that fall within the `m` to `n` range.

Solution Approach

The solution's approach can be divided into the following steps, each leveraging specific algorithms, data structures, or patterns commonly used in backtracking problems:

1. **Defining the Cross Matrix**: We start by defining a `cross` matrix where `cross[i][j]` represents the middle point (the point that should be previously visited) when drawing a line from dot `i` to dot `j`. This matrix is precomputed and hard-coded into the solution to comply with the constraint regarding the line segments.

2. **Backtracking Function `dfs`**: A recursive backtracking function `dfs` is implemented to explore all unique paths from any start dot. This function takes two parameters: the current dot `i` and a counter `cnt` that tracks the number of dots in the current pattern. The backtracking ends when the pattern's length exceeds `n`.

3. **Maintaining `vis` Vector**: We use a `vis` vector which is a list of boolean values indicating whether a dot has been visited (`True`) or not (`False`). This helps avoid reusing a dot in the same pattern, ensuring all dots are distinct.

4. **Exploring (DFS) and Counting**: For each dot, we perform a Depth-First Search (DFS) by recursively calling the `dfs` function on the next unvisited dot. If the current dot count is between `m` and `n` (inclusive), we increment our count of valid patterns (`ans`). The number of patterns that start from a certain dot can be obtained by simply returning `ans` at the end of exploring from that dot.

5. **Respecting the Constraints**: While exploring, we check if moving from the current dot `i` to the next dot `j` directly is valid or not by using the `cross` matrix. If it passes over a dot that hasn't been visited yet (`vis[x]` is `False` where `x` is the cross point), then we skip this path.

6. **Symmetric Property Exploitation**: The solution uses symmetry in the grid by calling the `dfs` function for only three distinct starting points - a corner, an edge midpoint, and the center. The counts for the corner and edge midpoints are then multiplied by 4 because there are four symmetric instances for corners and edge midpoints on the grid. The center is considered only once as it is unique.

7. **Computing the Final Result**: The final answer is the sum of patterns starting from the corner (returned value from `dfs` multiplied by 4), patterns starting from an edge midpoint (also multiplied by 4), and patterns starting from the center.

By following this approach, we systematically explore the entire search space while diligently sticking to the problem's rules and exploiting the grid's symmetric property to avoid redundant computation. This results in a computationally efficient solution to the problem.

Example Walkthrough

Let's take a simplified example to illustrate the solution. Imagine we are interested in finding valid patterns that consist of exactly 2 dots on the `3 x 3` grid. For simplicity, we'll start with a single starting point, which is the top-left corner `dot 1`. Here's how the backtracking process with the aforementioned approach looks like:

1. We begin by defining the `cross` matrix with NULL values indicating no cross points. Since our example only considers patterns with 2 dots, we don't need to worry much about cross points here.

2. The backtracking function `dfs` starts with `dot 1`. We mark `dot 1` as visited in the `vis` vector.

3. We call `dfs` with the starting `dot 1` and `cnt` initialized to 1 since we've already included `dot 1` in our pattern.

4. From `dot 1`, we explore the next possible dots. Let's try to move to `dot 2`. Since it has not been visited and does not require passing through a middle dot, we recurse with `dfs(2, cnt+1)`.

5. Inside `dfs(2, 2)`, we check if `cnt` equals `n` (which is 2 in our hypothetical case). As it does, we increment our pattern count `ans` by 1 for this valid sequence `[1, 2]`. There's no further recursion since we've reached the maximum dots allowed.

6. We backtrack and try the next possibility, moving to `dot 3`. Since it's valid and doesn't cross over any unvisited dots, we call `dfs(3, cnt+1)` and increment our count with the sequence `[1, 3]`.

7. We continue this process trying all possible dots (4, 5, 6, 7, 8, 9) and counting the number of valid patterns of length 2 starting with `dot 1`. For each valid move, we increment `ans`.

8. Since we're only calculating patterns starting from `dot 1` in this example, there's no need for symmetry exploitation. But in a full solution, we would multiply the `ans` from this step by 4 to account for patterns starting from corners (1, 3, 7, 9) due to the symmetric property.

9. The final answer for 2-dot patterns starting from the corner would be the value of `ans` obtained from all the recursive `dfs` calls starting with `dot 1`.

So, assuming our grid only allowed patterns to be exactly 2 dots starting from `dot 1`, the `ans` would represent the number of valid patterns, and the process highlighted here would enumerate them all. In a complete solution, patterns starting from different symmetrical points on the grid and of varying lengths between `m` and `n` would all be similarly explored to get the final count.

Python Solution

```python
class Solution:
    def numberOfPatterns(self, m: int, n: int) -> int:
        def dfs(current: int, count: int = 1) -> int:
            # If the count exceeds n, no further patterns can be formed
            if count > n:
                return 0

            # Mark the current number as visited
            visited[current] = True

            # Initialize answer as 1 if the current pattern length is within the range [m, n]
            patterns_count = int(m <= count)

            # Explore all possible next moves
            for next_num in range(1, 10):
                cross_num = cross[current][next_num]

                # Check if next_num is not visited and if there's no need to cross over a non-visited number
                if not visited[next_num] and (cross_num == 0 or visited[cross_num]):
                    patterns_count += dfs(next_num, count + 1)

            # Backtrack by marking current number as not visited
            visited[current] = False

            return patterns_count

        # Initialize a matrix that records the number that must be crossed to go from one number to another
        cross = [[0] * 10 for _ in range(10)]

        # Setting the cross numbers that need to be crossed between non-adjacent keys
        cross[1][3] = cross[3][1] = 2
        cross[1][7] = cross[7][1] = 4
        cross[1][9] = cross[9][1] = 5
        cross[2][8] = cross[8][2] = 5
        cross[3][7] = cross[7][3] = 5
        cross[3][9] = cross[9][3] = 6
        cross[4][6] = cross[6][4] = 5
        cross[7][9] = cross[9][7] = 8

        # Flags to keep track of visited numbers
        visited = [False] * 10

        # Start the patterns from numbers 1, 2, and 5
        # Multiply the return values by 4 for symmetrical positions (1,3,7,9) and (2,4,6,8)
        return dfs(1) * 4 + dfs(2) * 4 + dfs(5)

```

Java Solution

```java
class Solution {
    // Variable to store the minimum pattern length
    private int minPatternLength;
    // Variable to store the maximum pattern length
    private int maxPatternLength;
    // Matrix to store the jump-over number between two keys
    private int[][] crossOverPoints = new int[10][10];
    // Array to keep track of visited keys
    private boolean[] visited = new boolean[10];

    public int numberOfPatterns(int m, int n) {
        this.minPatternLength = m;
        this.maxPatternLength = n;
        // Initialize the crossover point between pairs of keys
        // where a third key has to be crossed over for a valid pattern
        crossOverPoints[1][3] = crossOverPoints[3][1] = 2;
        crossOverPoints[1][7] = crossOverPoints[7][1] = 4;
        crossOverPoints[1][9] = crossOverPoints[9][1] = 5;
        crossOverPoints[2][8] = crossOverPoints[8][2] = 5;
        crossOverPoints[3][7] = crossOverPoints[7][3] = 5;
        crossOverPoints[3][9] = crossOverPoints[9][3] = 6;
        crossOverPoints[4][6] = crossOverPoints[6][4] = 5;
        crossOverPoints[7][9] = crossOverPoints[9][7] = 8;
        // Calculate the number of valid patterns starting from different positions
        // Since the pattern is symmetric for positions 1, 3, 7, 9 and for positions 2, 4, 6, 8,
        // multiply the result of their respective DFS by 4 and add the count for position 5
        return dfs(1, 1) * 4 + dfs(2, 1) * 4 + dfs(5, 1);
    }

    // Helper method for depth-first search to find all valid patterns
    private int dfs(int position, int length) {
        // If current length exceeds the maximum pattern length, return 0 (as it's invalid)
        if (length > maxPatternLength) {
            return 0;
        }
        // Mark the current position as visited
        visited[position] = true;
        // If current length is within the pattern length bounds, increment the pattern count
        int count = (length >= minPatternLength) ? 1 : 0;
        // Explore all other keys as potential next steps in the pattern
        for (int nextKey = 1; nextKey < 10; ++nextKey) {
            int crossOverKey = crossOverPoints[position][nextKey];
            // If the next key hasn't been visited and (there is no crossover point or the crossover point has been visited)
            if (!visited[nextKey] && (crossOverKey == 0 || visited[crossOverKey])) {
                // Continue the depth-first search from the next key
                count += dfs(nextKey, length + 1);
            }
        }
        // Backtrack: unmark the current position to allow it to be part of other patterns
        visited[position] = false;
        // Return the total pattern count
        return count;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>
#include <cstring>

class Solution {
public:
    /* This function counts the number of distinct patterns from m to n moves. */
    int numberOfPatterns(int m, int n) {
        // cross are the numbers that need to be visited before visiting an index for a valid pattern
        int cross[10][10];
        std::memset(cross, 0, sizeof(cross));
        // visited array to keep track of visited numbers in the pattern
        bool visited[10];
        std::memset(visited, false, sizeof(visited));

        // Manually setting the cross over numbers
        cross[1][3] = cross[3][1] = 2;
        cross[1][7] = cross[7][1] = 4;
        cross[1][9] = cross[9][1] = 5;
        cross[2][8] = cross[8][2] = 5;
        cross[3][7] = cross[7][3] = 5;
        cross[3][9] = cross[9][3] = 6;
        cross[4][6] = cross[6][4] = 5;
        cross[7][9] = cross[9][7] = 8;

        // Depth-first search function to explore possible patterns
        std::function<int(int, int)> dfs = [&](int index, int count) {
            // If count exceeds n, stop exploring
            if (count > n) {
                return 0;
            }
            visited[index] = true; // Mark the index as visited
            // If current count between m and n, include it in answer
            int patterns = count >= m ? 1 : 0;
            // Explore remaining numbers
            for (int j = 1; j < 10; ++j) {
                int requiredBefore = cross[index][j];
                // Explore the next number if it has not been visited and the required number has been visited, if any
                if (!visited[j] && (requiredBefore == 0 || visited[requiredBefore])) {
                    patterns += dfs(j, count + 1);
                }
            }
            visited[index] = false; // Backtrack, mark index as unvisited
            return patterns; // Return the count of valid patterns
        };

        // Start DFS from 1, 2, 5 for symmetric patterns and sum up the results
        return dfs(1, 1) * 4 // Corners, so counted 4 times
             + dfs(2, 1) * 4 // Edges, so also counted 4 times
             + dfs(5, 1);    // Center, counted once
    }
};

```

Typescript Solution

```typescript
// Function to calculate the number of unique patterns that can be drawn on an Android lock screen.
// `m` is the minimum length of a pattern, `n` is the maximum length.
function numberOfPatterns(m: number, n: number): number {
    // Initialization of a matrix to track the crossing over number between two points.
    const cross: number[][] = Array.from({ length: 10 }, () => Array(10).fill(0));

    // Tracks whether a digit has been visited.
    const visited: boolean[] = Array(10).fill(false);

    // Define crossing over points which require to visit a middle point before reaching the destination.
    cross[1][3] = cross[3][1] = 2;
    cross[1][7] = cross[7][1] = 4;
    cross[1][9] = cross[9][1] = 5;
    cross[2][8] = cross[8][2] = 5;
    cross[3][7] = cross[7][3] = 5;
    cross[3][9] = cross[9][3] = 6;
    cross[4][6] = cross[6][4] = 5;
    cross[7][9] = cross[9][7] = 8;

    // Depth-First Search to explore all possible combinations from a starting digit.
    // `current` is the current digit being visited.
    // `count` is the current length of the pattern.
    const dfs = (current: number, count: number): number => {
        if (count > n) return 0;

        visited[current] = true;
        let patterns = 0;
        if (count >= m) {
            patterns++;
        }

        for (let next = 1; next < 10; ++next) {
            const requiredMiddlePoint = cross[current][next];
            if (!visited[next] && (requiredMiddlePoint === 0 || visited[requiredMiddlePoint])) {
                patterns += dfs(next, count + 1);
            }
        }

        visited[current] = false; // Backtrack
        return patterns;
    };

    // Calculate patterns starting from each of the corner keys (1, 3, 7, 9) and edges (2, 4, 6, 8), and the center (5).
    // Corners and edges are symmetric, so result is multiplied by 4.
    return dfs(1, 1) * 4 + dfs(2, 1) * 4 + dfs(5, 1);
}

```

Time and Space Complexity

Time Complexity

The provided code employs a Depth-First Search (DFS) strategy for traversing the space of all possible patterns that can be created on a 3×3 keypad, starting from digit `1`, digit `2`, and digit `5`. The worst-case time complexity would be calculated under the condition that we visit each digit exactly once and go through all possible patterns. Since there are 9 digits and each DFS invocation can lead up to 8 further invocations (minus the visited numbers and the cross numbers that require a middle number to be visited first), the upper bound can initially seem to be `O(9!)`. Yet, the time complexity is better due to the constraints that cut down the search space significantly - the `cross` matrix prevents jumps over unvisited keys.

Given that we invoke the DFS starting from 1, 2, and 5, these are our symmetrical starting points, and the answer will be the result from calling DFS from these start points, each multiplied by the number of symmetric equivalents on the keypad (corners are 4 times, edges are 4 times, and the center is unique):

1. Invoke `dfs(1)` and multiply the result by 4 (for the 4 corners).
2. Invoke `dfs(2)` and multiply the result by 4 (for the 4 edge middle points).
3. Invoke `dfs(5)` (the center).

Hence, we can simplify the naive factorial complexity as each corner and edge contribute equally.

Despite the upper bound given by the factorial, the actual time complexity is significantly less due to the constraints reducing the number of candidates at each step. However, providing an exact time complexity is difficult without empirical analysis or a more detailed combinatorial examination, which typically isn't expected for interview-style coding problems.

Space Complexity

The space complexity of the algorithm is `O(n)` due to the recursive nature of DFS, where `n` in this context is the maximum depth of the recursive stack, which corresponds to the maximum length of the pattern (up to 9). We also have additional data structures - `vis` (used to mark visited numbers) and `cross` (a matrix indicating the middle number that must be visited when going from one number to another in a direct line), but these are of fixed size and hence constitute `O(1)` additional space. Therefore, the space complexity remains `O(n)` where `n` stands for the number of recursive calls.