351. Android Unlock Patterns Medium Dynamic Programming Backtracking

Problem Description

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:

The problem belongs to the category of backtracking problems that simulate the process of constructing the unlock patterns of an

Leetcode Link

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.

solution to comply with the constraint regarding the line segments.

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

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.

not (False). This helps avoid reusing a dot in the same pattern, ensuring all dots are distinct.

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

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

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

by 4), patterns starting from an edge midpoint (also multiplied by 4), and patterns starting from the center.

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

starting points - a corner, an edge midpoint, and the center. The counts for the corner and edge midpoints are then multiplied by

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

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

1. We begin by defining the cross matrix with NULL values indicating no cross points. Since our example only considers patterns

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

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

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

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

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

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.

backtracking process with the aforementioned approach looks like:

with 2 dots, we don't need to worry much about cross points here.

passing through a middle dot, we recurse with dfs(2, cnt+1).

with dot 1. For each valid move, we increment ans.

def numberOfPatterns(self, m: int, n: int) -> int:

patterns_count = int(m <= count)</pre>

Explore all possible next moves

cross_num = cross[current][next_num]

for next_num in range(1, 10):

visited[current] = False

 $cross = [[0] * 10 for _ in range(10)]$

return patterns_count

cross[1][3] = cross[3][1] = 2

cross[4][6] = cross[6][4] = 5

cross[7][9] = cross[9][7] = 8

visited = [False] * 10

Flags to keep track of visited numbers

Start the patterns from numbers 1, 2, and 5

def dfs(current: int, count: int = 1) -> int:

If the count exceeds n, no further patterns can be formed

patterns_count += dfs(next_num, count + 1)

Backtrack by marking current number as not visited

starting with dot 1.

class Solution:

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

37

38

39

40

41

42

43

44

47

48

49

50

51

52

53

55

10

11

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

35

36

37

38

54 }

C++ Solution

1 #include <vector>

2 #include <functional>

#include <cstring>

class Solution {

public:

visited[position] = false;

int numberOfPatterns(int m, int n) {

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;

return 0;

std::memset(cross, 0, sizeof(cross));

std::memset(visited, false, sizeof(visited));

// If count exceeds n, stop exploring

// Explore remaining numbers

for (int j = 1; j < 10; ++j) {

// Manually setting the cross over numbers

int cross[10][10];

bool visited[10];

return count;

// Return the total pattern count

- 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**
- 5 if count > n: 6 return 0 8 # Mark the current number as visited visited[current] = True 9 10 11 # Initialize answer as 1 if the current pattern length is within the range [m, n]

Check if next_num is not visited and if there's no need to cross over a non-visited number

Initialize a matrix that records the number that must be crossed to go from one number to another

if not visited[next_num] and (cross_num == 0 or visited[cross_num]):

Setting the cross numbers that need to be crossed between non-adjacent keys

Multiply the return values by 4 for symmetrical positions (1,3,7,9) and (2,4,6,8)

// Backtrack: unmark the current position to allow it to be part of other patterns

/* This function counts the number of distinct patterns from m to n moves. */

// visited array to keep track of visited numbers in the pattern

// Depth-first search function to explore possible patterns

int requiredBefore = cross[index][j];

std::function<int(int, int)> dfs = [&](int index, int count) {

// cross are the numbers that need to be visited before visiting an index for a valid pattern

32 cross[1][7] = cross[7][1] = 433 cross[1][9] = cross[9][1] = 534 cross[2][8] = cross[8][2] = 535 cross[3][7] = cross[7][3] = 536 cross[3][9] = cross[9][3] = 6

```
45
             return dfs(1) * 4 + dfs(2) * 4 + dfs(5)
 46
Java Solution
    class Solution {
         // Variable to store the minimum pattern length
         private int minPatternLength;
         // Variable to store the maximum pattern length
         private int maxPatternLength;
  5
         // Matrix to store the jump-over number between two keys
  6
         private int[][] crossOverPoints = new int[10][10];
  8
         // Array to keep track of visited keys
         private boolean[] visited = new boolean[10];
  9
 10
 11
         public int numberOfPatterns(int m, int n) {
 12
             this.minPatternLength = m;
 13
             this.maxPatternLength = n;
 14
             // Initialize the crossover point between pairs of keys
 15
             // where a third key has to be crossed over for a valid pattern
 16
             crossOverPoints[1][3] = crossOverPoints[3][1] = 2;
 17
             crossOverPoints[1][7] = crossOverPoints[7][1] = 4;
 18
             crossOverPoints[1][9] = crossOverPoints[9][1] = 5;
             crossOverPoints[2][8] = crossOverPoints[8][2] = 5;
 19
 20
             crossOverPoints[3][7] = crossOverPoints[7][3] = 5;
 21
             crossOverPoints[3][9] = crossOverPoints[9][3] = 6;
 22
             crossOverPoints[4][6] = crossOverPoints[6][4] = 5;
 23
             crossOverPoints[7][9] = crossOverPoints[9][7] = 8;
 24
             // Calculate the number of valid patterns starting from different positions
 25
             // 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
 26
 27
             return dfs(1, 1) * 4 + dfs(2, 1) * 4 + dfs(5, 1);
 28
 29
 30
         // Helper method for depth-first search to find all valid patterns
         private int dfs(int position, int length) {
 31
 32
             // If current length exceeds the maximum pattern length, return 0 (as it's invalid)
             if (length > maxPatternLength) {
 33
                 return 0;
 35
 36
             // Mark the current position as visited
 37
             visited[position] = true;
             // If current length is within the pattern length bounds, increment the pattern count
 38
             int count = (length >= minPatternLength) ? 1 : 0;
 39
 40
             // Explore all other keys as potential next steps in the pattern
             for (int nextKey = 1; nextKey < 10; ++nextKey) {</pre>
 41
 42
                 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)
 43
                 if (!visited[nextKey] && (crossOverKey == 0 || visited[crossOverKey])) {
 44
                     // Continue the depth-first search from the next key
 45
                     count += dfs(nextKey, length + 1);
 46
```

29 if (count > n) { 30 31 32 visited[index] = true; // Mark the index as visited 33 // If current count between m and n, include it in answer 34 int patterns = count >= m ? 1 : 0;

```
39
                     if (!visited[j] && (requiredBefore == 0 || visited[requiredBefore])) {
                         patterns += dfs(j, count + 1);
 40
 41
 42
 43
                 visited[index] = false; // Backtrack, mark index as unvisited
 44
                 return patterns; // Return the count of valid patterns
 45
             };
 46
 47
             // Start DFS from 1, 2, 5 for symmetric patterns and sum up the results
             return dfs(1, 1) * 4 // Corners, so counted 4 times
 48
 49
                    + dfs(2, 1) * 4 // Edges, so also counted 4 times
 50
                    + dfs(5, 1); // Center, counted once
 51
 52 };
 53
Typescript Solution
  1 // Function to calculate the number of unique patterns that can be drawn on an Android lock screen.
  2 // `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);
  8
  9
 10
         // Define crossing over points which require to visit a middle point before reaching the destination.
         cross[1][3] = cross[3][1] = 2;
 11
         cross[1][7] = cross[7][1] = 4;
 12
 13
         cross[1][9] = cross[9][1] = 5;
 14
         cross[2][8] = cross[8][2] = 5;
 15
         cross[3][7] = cross[7][3] = 5;
         cross[3][9] = cross[9][3] = 6;
 16
         cross[4][6] = cross[6][4] = 5;
 17
 18
         cross[7][9] = cross[9][7] = 8;
 19
 20
         // Depth-First Search to explore all possible combinations from a starting digit.
 21
         // `current` is the current digit being visited.
 22
         // `count` is the current length of the pattern.
 23
         const dfs = (current: number, count: number): number => {
 24
             if (count > n) return 0;
 25
 26
             visited[current] = true;
 27
             let patterns = 0;
             if (count >= m) {
 28
 29
                 patterns++;
 30
 31
 32
             for (let next = 1; next < 10; ++next) {</pre>
 33
                 const requiredMiddlePoint = cross[current][next];
                 if (!visited[next] && (requiredMiddlePoint === 0 || visited[requiredMiddlePoint])) {
                     patterns += dfs(next, count + 1);
 35
 36
 37
 38
 39
             visited[current] = false; // Backtrack
 40
             return patterns;
         };
 41
 42
         // Calculate patterns starting from each of the corner keys (1, 3, 7, 9) and edges (2, 4, 6, 8), and the center (5).
 43
 44
         // Corners and edges are symmetric, so result is multiplied by 4.
 45
         return dfs(1, 1) * 4 + dfs(2, 1) * 4 + dfs(5, 1);
 46 }
 47
```

// Explore the next number if it has not been visited and the required number has been visited, if any

Time Complexity The provided code employs a Depth-First Search (DFS) strategy for traversing the space of all possible patterns that can be created

times, edges are 4 times, and the center is unique):

Time and Space Complexity

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

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.

from calling DFS from these start points, each multiplied by the number of symmetric equivalents on the keypad (corners are 4

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 0(9!). Yet, the time complexity is better due to the constraints that cut down the search space

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 0(1) additional space. Therefore, the space complexity remains 0(n) where n stands for the number of recursive calls.