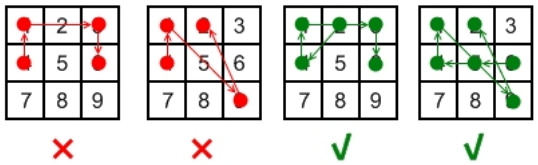
351. Android Unlock Patterns (/problems/android-unlock-patterns/)

June 9, 2016 | 6.6K views

Given an Android **3x3** key lock screen and two integers **m** and **n**, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of **m** keys and maximum **n** keys.

Rules for a valid pattern:

1. Each pattern must connect at least **m** keys and at most **n** keys.
2. All the keys must be distinct.
3. If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.
4. The order of keys used matters.



Explanation:

```
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
```

Invalid move: `4 - 1 - 3 - 6`
Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: `4 - 1 - 9 - 2`
Line 1 - 9 passes through key 5 which had not been selected in the pattern.

Valid move: `2 - 4 - 1 - 3 - 6`
Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern

Valid move: `6 - 5 - 4 - 1 - 9 - 2`
Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.

Example:

Given **m** = 1, **n** = 1, return 9.

Credits:

Special thanks to @elmirap (https://discuss.leetcode.com/user/elmirap) for adding this problem and creating all test cases.

Summary

After Android launched its "unlock pattern" system to protect our smart phones from unauthorized access, the most common question that comes to one's mind is: How secure exactly are these patterns? The current article gives an answer to this question, as presenting an algorithm, which computes the number of all valid pattern combinations. It is intended for intermediate users and introduces the following ideas: Backtracking, Arrays.

Solution

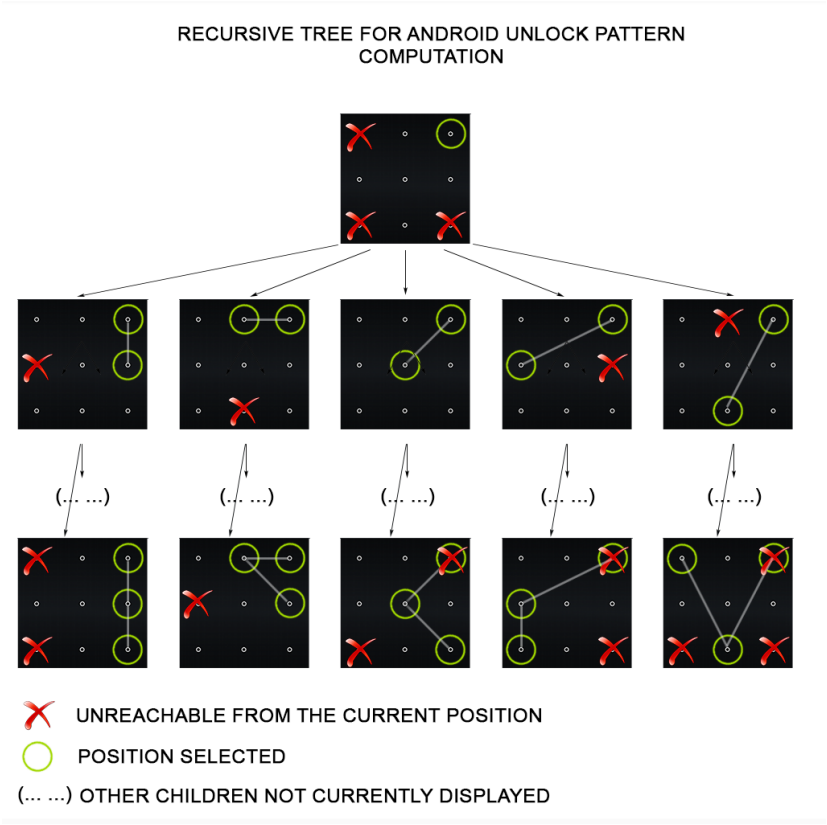
Approach #1: (Backtracking) [Accepted]

Algorithm

The algorithm uses backtracking technique to enumerate all possible k combinations of numbers $[1 \ldots 9]$ where $m \leq k \leq n$. During the generation of the recursive solution tree, the algorithm cuts all the branches which lead to patterns which doesn't satisfy the rules and counts only the valid patterns. In order to compute a valid pattern, the algorithm performs the following steps:

- Select a digit i which is not used in the pattern till this moment. This is done with the help of a $used$ array which stores all available digits.
- We need to keep last inserted digit $last$. The algorithm makes a check whether one of the following conditions is valid.
    - There is a knight move (as in chess) from $last$ towards i or $last$ and i are adjacent digits in a row, in a column. In this case the sum of both digits should be an odd number.
    - The middle element mid in the line which connects i and $last$ was previously selected. In case i and $last$ are positioned at both ends of the diagonal, digit mid = 5 should be previously selected.
    - $last$ and i are adjacent digits in a diagonal

In case one of the conditions above is satisfied, digit i becomes part of partially generated valid pattern and the algorithm continues with the next candidate digit till the pattern is fully generated. Then it counts it. In case none of the conditions are satisfied, the algorithm rejects the current digit i, backtracks and continues to search for other valid digits among the unused ones.


RECURSIVE TREE FOR ANDROID UNLOCK PATTERN COMPUTATION
(... ...)
(... ...)
(... ...)
(... ...)
(... ...)
UNREACHABLE FROM THE CURRENT POSITION
POSITION SELECTED
(... ...) OTHER CHILDREN NOT CURRENTLY DISPLAYED

Java

```
public class Solution {

    private boolean used[] = new boolean[9];

    public int numberOfPatterns(int m, int n) {
        int res = 0;
        for (int len = m; len <= n; len++) {
            res += calcPatterns(-1, len);
            for (int i = 0; i < 9; i++) {
                used[i] = false;
            }
        }
        return res;
    }

    private boolean isValid(int index, int last) {
        if (used[index])
            return false;
        // first digit of the pattern
        if (last == -1)
            return true;
        // knight moves or adjacent cells (in a row or in a column)
        if ((index + last) % 2 == 1)
            return true;
        // indexes are at both end of the diagonals for example 0,0, and 8,8
        int mid = (index + last)/2;
        if (mid == 4)
            return used[mid];
        // adjacent cells on diagonal  - for example 0,0 and 1,0 or 2,0 and //1,1
        if ((index%3 != last%3) && (index/3 != last/3)) {
            return true;
        }
        // all other cells which are not adjacent
        return used[mid];
    }

    private int calcPatterns(int last, int len) {
        if (len == 0)
            return 1;
        int sum = 0;
        for (int i = 0; i < 9; i++) {
            if (isValid(i, last)) {
                used[i] = true;
                sum += calcPatterns(i, len - 1);
                used[i] = false;
            }
        }
        return sum;
    }
}
```

Complexity Analysis

- Time complexity : $O(n!)$, where n is maximum pattern length

  The algorithm computes each pattern once and no element can appear in the pattern twice. The time complexity is proportional to the number of the computed patterns. One upper bound of the number of all possible combinations is :

$$\sum_{i=m}^{n} {}_9P_i = \sum_{i=m}^{n} \frac{9!}{(9-i)!}$$

- Space complexity : $O(n)$, where n is maximum pattern length In the worst case the maximum depth of recursion is n. Therefore we need $O(n)$ space used by the system recursive stack

Further Thoughts

The algorithm above could be optimized if we consider the symmetry property of the problem. We notice that the number of valid patterns with first digit 1, 3, 7, 9 are the same. A similar observation is true for patterns which starts with digit 2, 4, 6, 8. Hence we only need to calculate one among each group and multiply by 4.

You can find the optimized solution here (https://leetcode.com/discuss/104500/java-solution-with-clear-explanations-and-optimization-81ms).

Analysis written by: @elmirap.

Rate this article:
(/ratings/107/50/?return=/articles/android-unlock-patterns/) (/ratings/107/50/?return=/articles/an

Previous (/articles/coin-change/) Next (/articles/best-time-buy-and-sell-stock/)

Join the conversation

Login to Reply

Z (https://discuss.leetcode.com/user/zhangjunfly)

zhangjunfly commented 2 months ago

// indexes are at both end of the diagonals for example 0,0, and 8,8
int mid = (index + last)/2;
if (mid == 4)

isn't it 5 instead of 4?

L (https://discuss.leetcode.com/user/lesscode)

lesscode commented 4 months ago

we don't need to loop through m to n and compose sequence for each pattern length.
Instead we can compose just one time for pattern length n, and when the sequence length grows between m and n, we count it to sum.

M (https://discuss.leetcode.com/user/mitchellhe)

MitchellHe commented 9 months ago

I think the time cost of calcPatterns() is O(n!), and the time cost for the whole problem is O(n*n!).

N (https://discuss.leetcode.com/user/newtt)

newtt commented last year

the 8,8 in the comment is so confusing, please change it to (2,2)

S (https://discuss.leetcode.com/user/skysbjdy)

skysbjdy commented last year

indexes are at both end of the diagonals for example 0,0, and 8,8
adjacent cells on diagonal - for example 0,0 and 1,0 or 2,0 and //1,1
What do the examples mean ?? I don't get it.

View original thread (https://discuss.leetcode.com/topic/166)