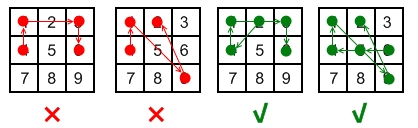
**Android Unlock Patterns**

Question

Android devices have a special lock screen with a 3 x 3 grid of dots. Users can set an "unlock pattern" by connecting the dots in a specific sequence, forming a series of joined line segments where each segment's endpoints are two consecutive dots in the sequence. A sequence of k dots is a **valid** unlock pattern if both of the following are true:

* All the dots in the sequence are **distinct**.
* If the line segment connecting two consecutive dots in the sequence passes through any other dot, the other dot **must have previously appeared** in the sequence. No jumps through non-selected dots are allowed.

Here are some example valid and invalid unlock patterns:



* The 1st pattern [4,1,3,6] is invalid because the line connecting dots 1 and 3 pass through dot 2, but dot 2 did not previously appear in the sequence.
* The 2nd pattern [4,1,9,2] is invalid because the line connecting dots 1 and 9 pass through dot 5, but dot 5 did not previously appear in the sequence.
* The 3rd pattern [2,4,1,3,6] is valid because it follows the conditions. The line connecting dots 1 and 3 meets the condition because dot 2 previously appeared in the sequence.
* The 4th pattern [6,5,4,1,9,2] is valid because it follows the conditions. The line connecting dots 1 and 9 meets the condition because dot 5 previously appeared in the sequence.

Given two integers m and n, return the ***number of unique and valid unlock patterns*** of the Android grid lock screen that consist of ***at least*** m keys and ***at most*** n keys.

Two unlock patterns are considered **unique** if there is a dot in one sequence that is not in the other, or the order of the dots is different.

**Example 1:**

**Input:** m = 1, n = 1

**Output:** 9

**Example 2:**

**Input:** m = 1, n = 2

**Output:** 65

**Constraints:**

* 1 <= m, n <= 9

## **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*k* combinations of numbers [1\dots 9][1…9] where m \leq k \leq n*m*≤*k*≤*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*i* which is not used in the pattern till this moment. This is done with the help of a used*used* array which stores all available digits.
* We need to keep last inserted digit last*last*. The algorithm makes a check whether one of the following conditions is valid.
  + There is a knight move (as in chess) from last*last* towards i*i* or last*last* and i*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*mid* in the line which connects i*i* and last*last* was previously selected. In case i*i* and last*last* are positioned at both ends of the diagonal, digit mid*mid* = 5 should be previously selected.
  + last*last* and i*i* are adjacent digits in a diagonal

In case one of the conditions above is satisfied, digit i*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*i*, backtracks and continues to search for other valid digits among the unused ones.

#### 

#### Coding Solution

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!)*O*(*n*!), where n*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} {\_9}P\_i = \sum\_{i=m}^{n} \frac{9!}{(9 - i)!}∑*i*=*mn*​9​*Pi*​=∑*i*=*mn*​(9−*i*)!9!​

* Space complexity : O(n)*O*(*n*), where n*n* is maximum pattern length In the worst case the maximum depth of recursion is n*n*. Therefore we need O( n)*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/problems/android-unlock-patterns/discuss/82463/Java-DFS-solution-with-clear-explanations-and-optimization-beats-97.61-(12ms)).

