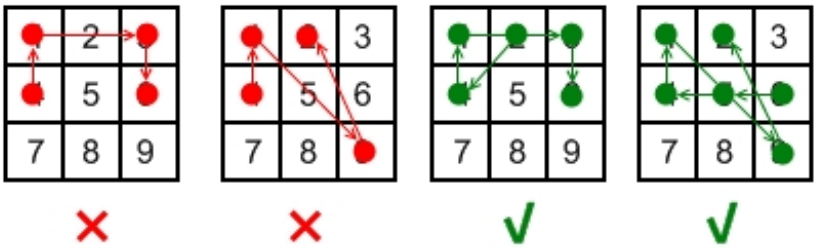
Android Unlock Patterns

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 for adding this problem and creating all test cases.

Solution 1

The optimization idea is that 1,3,7,9 are symmetric, 2,4,6,8 are also symmetric. Hence we only calculate one among each group and multiply by 4.

```
public class Solution {
    // cur: the current position
    // remain: the steps remaining
    int DFS(boolean vis[], int[][] skip, int cur, int remain) {
        if(remain < 0) return 0;
        if(remain == 0) return 1;
        vis[cur] = true;
        int rst = 0;
        for(int i = 1; i <= 9; ++i) {
            // If vis[i] is not visited and (two numbers are adjacent or skip num
ber is already visited)
            if(!vis[i] && (skip[i][cur] == 0 || (vis[skip[i][cur]]))) {
                rst += DFS(vis, skip, i, remain - 1);
            }
        }
        vis[cur] = false;
        return rst;
    }

    public int numberOfPatterns(int m, int n) {
        // Skip array represents number to skip between two pairs
        int skip[][] = new int[10][10];
        skip[1][3] = skip[3][1] = 2;
        skip[1][7] = skip[7][1] = 4;
        skip[3][9] = skip[9][3] = 6;
        skip[7][9] = skip[9][7] = 8;
        skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip[3][7] = skip[7]
[3] = skip[4][6] = skip[6][4] = 5;
        boolean vis[] = new boolean[10];
        int rst = 0;
        // DFS search each length from m to n
        for(int i = m; i <= n; ++i) {
            rst += DFS(vis, skip, 1, i - 1) * 4;    // 1, 3, 7, 9 are symmetric
            rst += DFS(vis, skip, 2, i - 1) * 4;    // 2, 4, 6, 8 are symmetric
            rst += DFS(vis, skip, 5, i - 1);        // 5
        }
        return rst;
    }
}
```

written by zzz1322 original link here

Solution 2

The general idea is DFS all the possible combinations from 1 to 9 and skip invalid moves along the way.

We can check invalid moves by using a jumping table. e.g. If a move requires a jump and the key that it is crossing is not visited, then the move is invalid. Furthermore, we can utilize symmetry to reduce runtime, in this case it reduced from ~120ms to ~70ms.

I felt clueless when first encountered this problem, and considered there must be lots of edge cases. Turns out, it's pretty straight forward. Hope this solution helps :D

```
private int[][] jumps;
private boolean[] visited;

public int numberOfPatterns(int m, int n) {
    jumps = new int[10][10];
    jumps[1][3] = jumps[3][1] = 2;
    jumps[4][6] = jumps[6][4] = 5;
    jumps[7][9] = jumps[9][7] = 8;
    jumps[1][7] = jumps[7][1] = 4;
    jumps[2][8] = jumps[8][2] = 5;
    jumps[3][9] = jumps[9][3] = 6;
    jumps[1][9] = jumps[9][1] = jumps[3][7] = jumps[7][3] = 5;
    visited = new boolean[10];
    int count = 0;
    count += DFS(1, 1, 0, m, n) * 4; // 1, 3, 7, 9 are symmetrical
    count += DFS(2, 1, 0, m, n) * 4; // 2, 4, 6, 8 are symmetrical
    count += DFS(5, 1, 0, m, n);
    return count;
}

private int DFS(int num, int len, int count, int m, int n) {
    if (len >= m) count++;
    len++;
    if (len > n) return count;
    visited[num] = true;
    for (int next = 1; next <= 9; next++) {
        int jump = jumps[num][next];
        if (!visited[next] && (jump == 0 || visited[jump])) {
            count = DFS(next, len, count, m, n);
        }
    }
    visited[num] = false; // backtracking
    return count;
}
```

written by david.liu original link here

Solution 3

If we use the symmetry, we can only start from 1, 2 and 5 then multiply the results of 1 and 2 by 4. (170ms) Thanks to @woaizuguo999

```
int res=0;
public int numberOfPatterns(int m, int n) {
    boolean[][] keyboard = new boolean[3][3];
    int ret=0;
    for (int p=m;p<=n;p++){
        for (int i=0;i<2;i++){
            for (int j=0;j<2;j++){
                if (j == 0 && i == 1) continue;
                keyboard[i][j] = true;
                helper(keyboard,p-1,i,j);
                keyboard[i][j] = false;
                ret += (i == 1 && j == 1)? res:4*res;
                res=0;
            }
        }
    }
    return ret;
}
private void helper(boolean[][] keyboard,int left, int x, int y){
    if (left == 0){
        res++;
        return;
    }
    for (int i=0;i<3;i++){
        for (int j=0;j<3;j++){
            if (keyboard[i][j]
                ||  (x==i && Math.abs(y-j)>1) && !keyboard[x][1]
                ||  (y==j && Math.abs(x-i)>1) && !keyboard[1][y]
                ||  (x+y == i+j) && Math.abs(x-i) >1 && !keyboard[1][1]
                ||  (x-y == i-j) && Math.abs(x-i) >1 && !keyboard[1][1]
                ||  (x == i && y == j)) {
                continue;
            }
            else{
                keyboard[i][j] = true;
                helper(keyboard,left-1,i,j);
                keyboard[i][j] = false;
            }
        }
    }
}
```

And we can continue improving the performance by using symmetry in step 1, which is the next step after start. For start from 1, only consider 2 6 and 5. For start from 2, only consider 3,6,9 and 5. For start from 5, only consider 1 and 2.(72 ms)

```
int res=0;
public int numberOfPatterns(int m, int n) {
    boolean[][] keyboard = new boolean[3][3];
    int ret=0;
```

```
    for (int p=m;p<=n;p++){
        for (int i=0;i<2;i++){
            for (int j=0;j<2;j++){
                if (j == 0 && i == 1) continue;
                keyboard[i][j] = true;
                helper(keyboard,p-1,i,j,true);
                keyboard[i][j] = false;
                ret += (i == 1 && j == 1)? res:4*res;
                res=0;
            }
        }
    }
    return ret;
}

private void dfshelper(boolean[][] keyboard, int left, int x, int y){
    keyboard[x][y] = true;
    helper(keyboard,left,x,y,false);
    keyboard[x][y] = false;
}

private void helper(boolean[][] keyboard,int left, int x, int y, boolean step1){
    if (left == 0){
        res++;
        return;
    }
    if (step1){
        if (x == 0 && y == 0){
            dfshelper(keyboard,left-1,0,1);
            dfshelper(keyboard,left-1,1,2);
            int temp = 2*res;
            res = 0;
            dfshelper(keyboard,left-1,1,1);
            res += temp;
        }
        if (x == 0 && y == 1){
            dfshelper(keyboard,left-1,0,2);
            dfshelper(keyboard,left-1,1,2);
            dfshelper(keyboard,left-1,2,2);
            int temp = 2*res;
            res = 0;
            dfshelper(keyboard,left-1,1,1);
            res += temp;
        }
        if (x == 1 && y == 1){
            dfshelper(keyboard,left-1,0,0);
            dfshelper(keyboard,left-1,0,1);
            res=res*4;
        }
    }
    else{
        for (int i=0;i<3;i++){
            for (int j=0;j<3;j++){
                if (keyboard[i][j]
                    ||  (x==i && Math.abs(y-j)>1) && !keyboard[x][1]
                    ||  (y==j && Math.abs(x-i)>1) && !keyboard[1][y]
                    ||  (x+y == i+j) && Math.abs(x-i) >1 && !keyboard[1][1]
```

```
                    ||  (x+y == i+j) && Math.abs(x-i) >1 && !keyboard[1][1]
                    ||  (x-y == i-j) && Math.abs(x-i) >1 && !keyboard[1][1]
                    ||  (x == i && y == j)) {
                    continue;
                }
                else{
                    keyboard[i][j] = true;
                    helper(keyboard,left-1,i,j,false);
                    keyboard[i][j] = false;
                }
            }
        }
    }
}
```

written by qwe5103368 original link here

From Leetcoder.