Security of Android Unlock Pattern

Tong Wu, Xun Tang, Xinyu Lyu
Department of Electrical and Computer Engineering
Rutgers University
{*tw445,xt63, xl422*}*@scarletmail.rutgers.edu*

May 6, 2018

Abstract

Android unlock patterns are commonly used in Android devices. It may look complicated and hard to crack. In this paper, we investigated the security of this type of unlock system. We considered the number of gestures for each number of used vertices as a target of security. It is difficult to count out how many gestures it will be because of the valid rules. We explored 5 different algorithms to solve the problem. With implementation and analysis, we compared the performance.

Keywords: Android Unlock Pattern; Depth First Search; Back Tracking and DP.

1 Introduction

1.1 Background

Originally introduced in 2008, Android's lock pattern screen was presented as both an easier and more secure alternative to traditional numeric passcodes. In other words, this system is a series of gesture to unlock your Android smartphones. Normally, the Android unlock pattern has 9 slots on the screen organized in a 3×3 matrix. To unlock the phone a so called pattern has to be drawn on the screen, which means connecting certain points in a certain order.

1.2 Motivation

In the common unlock method using 4 digits as password, which has exactly 10000 different combinations. But after Android launched this kind of Unlock Pattern system to protect our smartphones from unauthorized access, the most common question that comes to ones mind is: "How secure exactly are these patterns?". In order to figure out

this question, in the paper, we intend to analyze the total number of possible combinations of this unlock pattern by using 5 feasible algorithms: Brute Forth, DFS, BackTracking, DP and A smart way. Then we compared the performance among these 5 algorithms.

1.3 Valid Gestures

Before we head into the algorithms, it is necessary for us to clarify what specific gestures are accepted by this system. Basically, given an Android 3×3 key lock screen, a valid unlock gesture must contain at least 2 but no more than 9 plots. The gesture must be consecutive and the keys must be distinct. In each valid gesture, the visited plot cannot be visited twice, except that it can be covered in a routine and get visited again. Additionally, every plot in the jumping curve must be marked as a visited plot. And the orders of plots used matters. We give some valid and invalid gestures below for convenience.

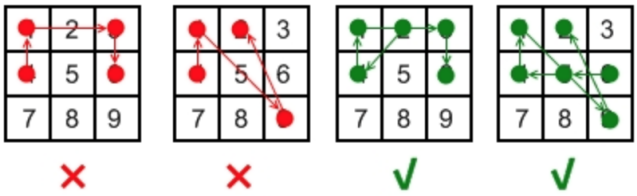

Figure 1:
Invalid move: 4 - 1 - 3 - 6
Line 1 - 3 passes through key 2 which had not been selected in the pattern.
Invalid move: 4 - 1 - 9 - 2
Line 1 - 9 passes through key 5 which had not been selected in the pattern.
Valid move: 2 - 4 - 1 - 3 - 6
Line 1 - 3 is valid because it passes through key 2, 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.

2 Implementation

2.1 Infrastructure

To test the performance, we used the following machine:
Intel Core $i5-5257U$ with 8GB memory and 64 bit build Late 2015 MacBook Pro. The inversion of Java is $1.8.0_144-b01$.

2.2 Brute Forth

The straight way is finding the invalid gestures and using total number of gestures minus the number of invalid gestures. In this method, vertices are considered as numbers 1 to 9. Therefore, the total number of gestures could be calculated by the permutation of N. (where N is the number of vertices used in gesture)

2.2.1 Permutation

In mathematics, the notion of permutation relates to the act of arranging all the members of a set into some sequence or order, or if the set is already ordered, rearranging (reordering) its elements, a process called permuting[1]. There are 2 ways of permutation calculation. The first one is a recursive way and the other one is based on Alphabetical order. The other is based on Alphabetical sort algorithm.

The recursive method

The idea of this method is using recursion to select the next number of order. When we calculate the permutation of N, we try each number in ascending order. Every step we add a number to the current permutation, we go to next unused number and do the recursion. After recursion, we step back to the last stage and mark this number unused again. It has a similar idea of Back Tracking algorithm.

```
public Permutation(int max) {
        this.max = max;
        array = new int[max + 1];
        hold = new int[max + 1];
    }

public void permute(int step) {
        ...
        for(int num = 1; num <= max; num++) {
            if(hold[num] == 0) {
                array[step] = num;
                hold[num] = 1;
                permute(step + 1);
                hold[num] = 0;
            }
        }
```

Alphabetical order method

Alphabetical order is a system whereby strings of characters are placed in order based on the position of the characters in the conventional ordering of an alphabet. It is one of the methods of collation[2]. For this method, we define an order between two numbers in a given sequence of the number set. Based on this, we define the order between two permutations by comparing each number from first to last one.

2.2.2 Find the Invalid Gestures

The invalid gestures are presented as a stride over an unvisited vertice. A hashmap was used to maintain the circumstances of a stride over. We get the permutation from the previous 2.2.1 part. By traversing it, we test if the permutation items contain keys in the hashmap and if the items substring from 0 to index of key doesnt contain values in corresponding keys. If so, it means a valid gesture.

```
for (String item : list){
            boolean found = false;
            for (Map.Entry<String,String> entry : map.entrySet()){
                if (item.contains(entry.getKey()) && item.substring(0,item.
                    indexOf(entry.getKey())).contains(entry.getValue())) {
                    found = true;
                    break;
                }
            }
            if (!found) count++;
        }
```

Till now, we used permutation algorithm to get all situations of gestures. By using a hashmap, we find the valid gestures in these gestures.

2.2.3 Analysis

Obviously, the time complexity is huge due to many times of traversals. In permutation part, because we used recursion, the time complexity is $O(N!)$. Where N is the number of used vertices. In 2.2.2 part, the time complexity of traversal is $O(N!)$, and the complexity of checking hash map is $\sim 60N$. Therefore, the total time complexity is $\sim 60N * N!$.

2.3 DFS Implementation

2.3.1 DFS

Depth-First-Search(DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. We consider the nine points of the pattern as nine integers from 1 to 9.

2.3.2 Prerequisite

So how do we use DFS in this algorithm? Actually, the unlock pattern is a sort of directed graph which indicates, for instance, 2 to 3 is different from 3 to 2. We can implement DFS from one start point (root node) to the end point to find a path, and the length of the path differs from 2 to 9. The general idea is to DFS all the possible combinations from 1 to 9 and skip invalid moves along the way.

```
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;
```

This means when you want to use a gesture, e.g., from 1 to 3 or 3 to 1, 2 must have been visited. 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.

2.3.3 Core algorithm

```
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] && (skip[cur][i] == 0 || (vis[skip[cur][i]]))) {
            rst += DFS(vis, skip, i, remain - 1);
        }
    }
    vis[cur] = false;
    return rst;}
```

At first, we declare two arrays and two variables. Array $vis[]$ is the boolean datatype, storing whether the vertex or the slot has been visited. We have mentioned array $skip[][]$ above. Variable cur means the current position that DFS is traversing. Variable remain means the steps remaining to get to the endpoint. In the if statement, when the number of remaining steps is only one, the DFS has reached the last point of the path, then return one. The current position will be marked as visited. Then a for loop will be implemented. This loop starts from 1 to 9 which means consider each point as the start point to do the DFS. And in this loop, there exists a constraint condition to remove the invalid gestures. When DFS traverse from one vertex to another, if the vertex hasnt been visited, and the two slots can be directly connected (no need to skip, $skip[][] = 0$) or the slot between two slots has already been visited, then the DFS can execute successfully. After the loop, mark current position unvisited.

2.3.4 Optimization

Furthermore, we can utilize symmetry to reduce runtime. 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.
It is a huge optimization because the number of calculations are hugely reduced.

```
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
}
```

2.3.5 Analysis

The most part of this algorithm is implementing DFS, the time complexity of DFS is $O(E + V)$. Basically, E means edges, V means vertices. However, technically, the time complexity of this algorithm can not be accurate. Because every time a for loop is implemented, the edges and vertices may be different, so it causes the complexity of the algorithm to be different from $O(E + V)$. Also, it contains some recursive procedures, which leads to another deviation. But it doesnt matter this algorithm to be an efficient algorithm to compute the question.

2.4 Back Tracking Implementation

2.4.1 Back Tracking Algorithm

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.[3]

2.4.2 Core idea

In this problem, the algorithm uses backtracking technique to enumerate all possible k combinations of numbers [1 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.

2.5 DP Implementation

2.5.1 The key functions

```
void calcDP()
  loop to set the base starting keys for further search with length as 1
  loop to visit dp matrix with all the length parmeters, length++
    loop to visit dp matrix with all the visited states, states ++
      loop to visit dp matrix with all the endNodes, endNode ++
        judge if exists the pattern with the given state, originally
            develop from the base starting keys
          loop to visit all the potential next keys
            judge if the potential next keys visited before
              update the visited state
              judge if the next key is valid
                if valid, update dp[len + 1] [nextState][i]  with the dp[
                    len][state][endNode]
```

2.5.2 The general idea for DP

The DP implementation is the most elusive method with the least time complexity. And it uses the $dp[10][1 << 9][10]$ as an android pattern table to store all the potential patterns. Use $calcDP()$ functions to find out the valid gestures. Last, in the $numberOfPatterns(m, n)$ function to sum up the total number of valid patterns.

2.5.3 The preparatory work

First, we initialize a two-dimensional matrix $skip[10][10]$ to store the middle keys between key-key pairs, i.e., $skip[1][3] = 2$ means key 2 is the middle key between key 1 and key 3. Then we initialize a three-dimensional matrix $dp[10][1 << 9][10]$ as an android pattern table to store all the potential android patterns. And the $dp[len][state][endnode]$indicates the number of patterns with length len, visited nodes equal to $state$ and the last node in the pattern is $endNode$.

2.5.4 The calculated strategy

The general idea of $calDP()$:

We use $calcDP()$ functions to find out all the valid gestures and save as the values of the elements in $dp[10][1 << 9][10]$. For the $calcDP()$ function, it uses the Dynamic programming together with the bitmask method to save the valid android patterns in the $dp[10][1 << 9][10]$.

Construction of $calDP()$:

The $calcDP()$ function mainly consist of five loops as follows:

- The first loop set all the valid android patterns with length(1). Also, the first loop set up the base visited states for the further search. And there are nine of them, for they visit only one point. And in the second/third/fourth loops, it visits all the elements of the potential gestures in the android pattern table $dp[10][1 << 9][10]$.
- And in the second/third/fourth loops, it visits all the elements of the potential gestures in the android pattern table $dp[10][1 << 9][10]$. In the second loop, it increases the length from 1 to 9.
- For the third loop, we initialize a 9-bit bitmask state to store the visited keys and visit all the potential states.
- In the fourth loop, we traverse all the elements with possible $endNodes$ from 1 to 9.
- In the last loop, we set parameter i representing the potential next element of the android pattern array that you could visit.

Find the valid Gestures of calDP():

Before the last step of the calcDP() function, we need to figure that if the next key is valid or not. Therefore, we need to find out whether there is a middle key between the previous key and the next chosen key, with this statement $skip[endNode][i] == 0$. And if there is a middle key, we need to find out whether the middle key has been visited before.

Update the new state of calDP():

In the last step of the $calcDP()$ function, when through all the judgements, we could update $dp[len + 1][nextState][i]$ with the value of $dp[len][state][endNode]$. $[Len + 1]$ is because we have visited one more valid keys, the length will grow by adding 1. And we update the current $[state]$ with the $[nextState]$, for we have visited one more keys in the android pattern.

2.5.5 Performance

Time Complexity

The time complexity of the DP-bitmask method achieves its smallest, among these five methods which are $O(2N)$. Here we take the maximal length of the android pattern is N. When traversing all the elements in the $dp[len][state][endNode]$ array, it costs about $100N * 2^{11}$ times of access to the dp and $skip$ array. Therefore the time complexity is $100N * 2^{11}$.

Space Complexity

The space complexity is enormous, for it initializes a three-dimensional matrix $dp[10][1 << 9][10]$. It costs about $100 * 2^9$. Here we take the maximal length of the android pattern is N.

2.6 A Smarter Way

2.6.1 The general idea

The smarter method uses a smarter way to consider the positional relationship between the current plot and the valid plots.

2.6.2 The key function

```
int count(m,n,used plots,last plot, next plot)
  loops to visit all the next nine plots on the android pattern
        Add up x and y coordinates of  last plot and next plot together,
            update used
            judge whether the next lot is valid to visit
                    recursively call count(m-1,n-1, used2, last plot next
                        plot) to add up the number of the valid android
                        gestures
```

2.6.3 The preparatory work

First, we could initialize a 9-bit bitmask named used telling which keys have already been used. Then we set the two-dimensional coordinate (i1, j1) for the previous key and (i2, j2) for the next key.

2.6.4 The recursion method

We start from the plot (1,1), with the previous plot as (1,1) which is the starting point. After get the number, we recursively invoke the count(m-1,n-1,i2,j2,i,j) function again. In this time, we use the plot(i2,j2) as the last plot, and the (i,j) as the next valid plot, based on the *used*2 as the visited plots. Here, we use m-1 and n-1 to call the recursion, until n decreases to 0. It means that the pattern needs to visit at most 0 plot so that the return would be 0. And in the recursion, we could find that both m and n minus 1 in next function call. And when m is smaller than 0, the initialized parameter number should be 1. It means that the situation that pattern with length as 1, starting from (i1,j1) to (i2,j2), i.e., the pattern only with (1,1) or (2,2) and so on. Repeating the above statement of the recursion function, we could finally get the total valid gesture.

2.6.5 Find the valid Gestures

The general idea:

When we are trying to find the next valid keys from the previous one, we need to judge whether there is a key in the middle of such two keys. If there is not a middle key, it is valid to move to that next key. If there is a middle key, you need to find out whether it has been visited before. And if it has not been visited before, it is not a valid next key. It has been visited before; it is valid to move to that next key crossing the middle key.

Implementation

- To implement the judgment that if it is valid to move to the chosen next key, we initialize two parameters I and J. $I = i2 + i$ mean that add up the x-coordinates of the previous key and the next chosen key. $J = j2 + j$ means that add up the y-coordinates of the previous key and the next chosen key.

- And (i, j) means the coordinates of the next key, which starts from (0,0) to (2,2) corresponding to the key at the top left corner and the bottom right corner. And if $I\%2 == 1$, it means there are no middle keys between such two keys, i.e., from (0,0) to (1,1). And $(I\%2) == 1||(J\%2) == 1$ contains all the valid next keys without the middle keys.

- To judge if the key has been visited before, we use to update the 9-bit bitmask $used2$ to update the visited keys.

- Also, when we are judging whether the middle one has been visited before, we use this statement $used2\&(1 << (I/2 * 3 + J/2)))! = 0$. If it equals to 0, it means the middle has been visited, so it is valid to move to the next chosen key cross the middle key.

2.6.6 Performance

Time complexity

If there are X elements in the pattern, the complexity is: $O((9^n) * 9) = O(9^{n+1})$
(Where 9^n refers to 9 next steps from any given point and that's done n times. Multiplied by 9 because the beginning point can be any of 9 elements).

Space complexity

The space complexity is pretty small which is $O(1)$, for we only use a 9-bit bitmask to store the visited keys and an int number to store the result.

3 Benchmarks

3.1 Security result

In this problem, we want to test the security of Android unlock pattern. The main target is to calculate the number of valid gestures for each number of used vertices. The more gestures it has, the more secure it is and more difficult to crack it. The result is shown in Figure 2. We can see that with 8 or 9 vertices, Android unlock pattern has similar security with a 5-digits password which has 10^5 different possibilities. However, for most people, it is hard to create 8 or 9 vertices of gestures which are not easy to remember. Thus, gestures with 5 to 7 vertices are most common. In this case, the pattern has the same security with a 4-digits password like iPhone.

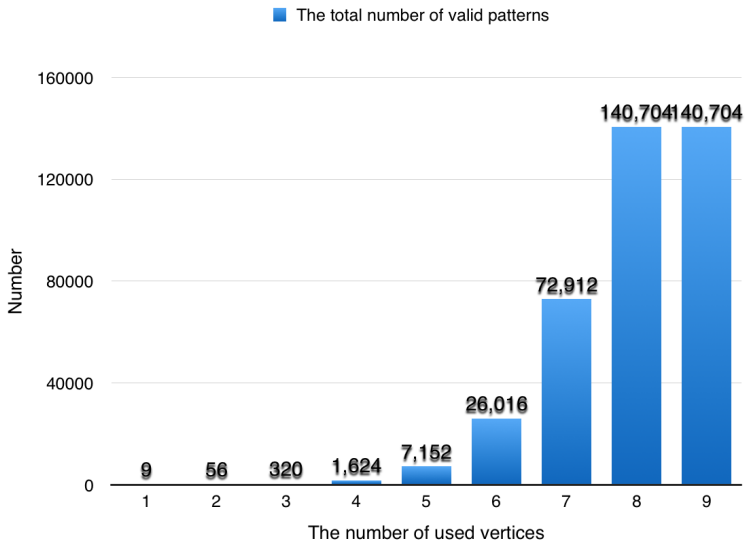

Figure 2: The security result on different number of used vertices. Gestures with less vertices have lower security and are easier to be cracked. But it is not the case when we use 8 and 9 vertices of gestures because they have same number of gestures.

3.2 Performance

After knowing the security of different gestures, the performances of algorithms we used are worth to test. We solved this problem on 5 algorithms and tested the total execution time. The results are shown in Figure 3. Due to the huge distance between BruteForth to others, the performances of other 4 algorithms are not clear. We built another figure with only 4 algorithms which is shown in Figure 4.

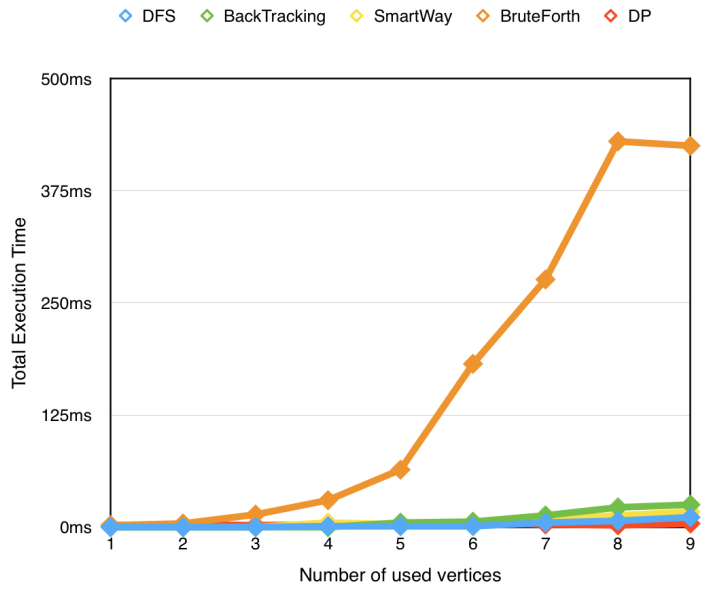

Figure 3: The BruteForth algorithm took the most time among five algorithms. And the differences between BruteForth with other 4 algorithms are huge. However, it is hard to find out the performance among different 4 algorithms in this figure. Therefore we built another Figure 4 which has deleted BruteForth algorithm to show the performance of left 4 algorithms.

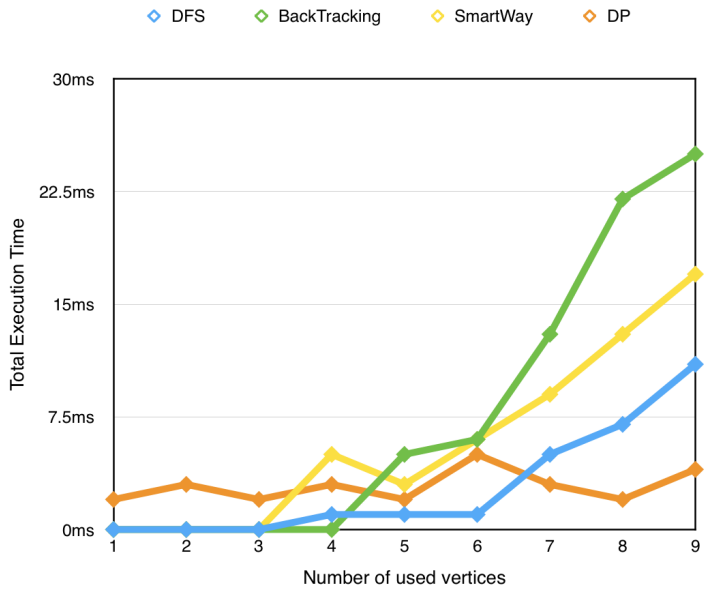

Figure 4: It is obvious that DP has a steady performance with some vertices grows up. But the execution times of other three algorithms increased with different velocities. When the number of vertices is small, such like 1 to 4, the DFS, BackTracking, and SmartWay have better performance than DP. However, with the number of vertices grows up, DP is the king.

4 Future Work

We found that the security of 3×3 unlock pattern is not high enough than we thought. In the future, we'd like to expand the size of pattern to 4×4 and even 5×5, which would be much more complicated.

5 Conclusion

In the paper, we introduced the unlock pattern of Android devices and explored the security of this unlock system. The number of valid gestures indicates the complexity of gesture password. Five algorithms were used to calculate the valid gestures with the different number of vertices.

The result of security test shows that an 8 or 9 vertices gesture has a security order of 10^5. That is equivalent to a 5-digits password, which is much securer than iPhone 4-digits unlock system. However, commonly used gestures use only 5 to 7 vertices, which has less than 10^4 order of security. It is not safe enough as people thought.

The performances of 5 algorithm implementations are also tested. Brute-forth has the worst performance in both execution time and memory spaces. DP algorithm has the least order of growth and has best performance. The smart way has the best space performance and doesn't use any extra memory. Thanks to the huge optimization of DFS, it has the best performance in the last 3 algorithms.

References

[1] Bna, Mikls (2004), Combinatorics of Permutations, Chapman Hall-CRC, ISBN 1-58488-434-7

[2] Daly, Lloyd. Contributions to the History of Alphabetization in Antiquity and the Middle Ages Brussels, 1967. p. 25

[3] Donald E. Knuth (1968). The Art of Computer Programming. Addison-Wesley.

[4] Dynamic Programming and Optimal Control by Dimitri P. Bertsekas ISBNs: 1-886529-26-4 (Vol. I, 4th Edition), 1-886529-44-2 (Vol. II, 4th Edition), 1-886529-08-6 (Two-Volume Set, i.e., Vol. I, 4th ed. and Vol. II, 4th edition)

[5] Dynamic Programming and Bit Maskinghttps: //www.hackerearth .com/zh/practice /algorithms/dynamic-programming/bit-masking/tutorial/

[6] DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS* ROBERT TARJAN"

[7] Complexity Metrics and User Strength Perceptions of the Pattern-Lock Graphical Authentication Method Panagiotis AndriotisTheo TryfonasGeorge Oikonomou