# Daily Coding Problem #239

## Problem

This problem was asked by Uber.

One way to unlock an Android phone is through a pattern of swipes across a `1-9` keypad.

For a pattern to be valid, it must satisfy the following:

- All of its keys must be distinct.
- It must not connect two keys by jumping over a third key, unless that key has already been used.

For example, `4 - 2 - 1 - 7` is a valid pattern, whereas `2 - 1 - 7` is not.

Find the total number of valid unlock patterns of length `N`, where `1 <= N <= 9`.

## Solution

Let's first try to solve the problem without any restrictions on jumping over keys. If there are `N` starting numbers to choose from, we will have `N - 1` options for the second number, `N - 2` options for the third, and so on. Mathematically we can recognize the product of these to be `N!`. But we can also use depth-first search to explore these patterns.

Each time we visit a number, we mark it as visited, traverse all paths starting with that number, and then remove it from the visited set. Along the way we keep a running count of the number of paths seen thus far, which we return as our result.

```
def num_paths(current, visited, n):
    if n - 1 == 0:
        return 1

    paths = 0
    for next_number in range(1, 10):
        if next_number not in visited:
            visited.add(next_number)
            paths += num_paths(next_number, jumps, visited, n - 1)
            visited.remove(next_number)

    return paths
```

To modify this to account for jumps, we can use a dictionary mapping pairs of keys to the key they skip over. Before visiting a number, we check to see that either the current and next number do not exist as a pair in this dictionary, or that their value has already been visited.

Notice also that because of symmetry, the number of patterns starting from `1` is the same as the number of patterns starting from 3, 7, and 9. For example, the path `1 - 6 - 3 - 8` can be rotated `180` degrees to get `9 - 4 - 7 - 2`. Similarly, paths starting with 2, 4, 6, and 8 are all rotationally symmetric. So our answer can be expressed as:

```
4 * num_paths(1) + 4 * num_paths(2) + 1 * num_paths(5)
```

Putting it all together, the code would look something like this:

```
def num_paths(current, jumps, visited, n):
    if n - 1 == 0:
        return 1
    paths = 0
    for next_number in range(1, 10):
        if next_number not in visited:
            if (current, next_number) not in jumps or jumps[(current, next_number)] in visited:
                visited.add(next_number)
                paths += num_paths(next_number, jumps, visited, n - 1)
                visited.remove(next_number)

    return paths

def unlock_combinations(n):
    jumps = {(1, 3): 2, (1, 7): 4, (1, 9): 5,
             (2, 8): 5,
             (3, 1): 2, (3, 7): 5, (3, 9): 6,
             (4, 6): 5, (6, 4): 5,
             (7, 1): 4, (7, 3): 5, (7, 9): 8,
             (8, 2): 5,
             (9, 1): 5, (9, 3): 6, (9, 7): 8}

    return 4 * num_paths(1, jumps, set([1]), n) + \
           4 * num_paths(2, jumps, set([2]), n) + \
           1 * num_paths(5, jumps, set([5]), n)
```

Even though the jump restrictions have limited the options at each next step, the time complexity for each of the three starting points is still `O(N!)`.