Backtracking 回溯

BackTracking 的题目比较典型，如果出现了，一定要做对。

LC489 robot room cleaner 待做啊

LC022

LC 基础 BackTracking 练习：

039 开拓思路，消除惯性。
040 039 相关有一点意思
216 039 相关 水笔
046 水笔
047 有点难
077 还行

2D BackTracking
037 数独是一个 2-D, 事实上比 N 皇后要复杂。
051 N-皇后也可以按 2-D 写。
079 很像 DFS，但是是 backtracking。 2-D

回溯法是什么?

回溯本质上是一种暴力法(Brute Force)。暴力法是遍历解空间(solution space)。
回溯分步去解决问题，如果发现当前无论如何不存在解的话，它会往回退一步甚至几步。

回溯特别适用于解决约束满足问题(constraint satisfaction problems+)，比如
纵横字谜，数独，八皇后等等。

回溯法比暴力法强的地方是我们还用到了剪枝(pruning)。
回溯不去遍历明显不可能的子空间，从而大大减少了时间复杂度。

回溯的解空间一般具有树结构，例如数独的解空间是 81 层的 10 叉树。而 N 皇后的
解空间是 N 层的 N 叉树。回溯通过 DFS 在解空间中寻找可能的解/解集，并根据问题
本身的特性使用剪枝/限界函数(pruning)来去掉不可能的 stem。如果在通过了剪枝的
可能的子空间里不能遍历到一个答案，那么就回退到上一步或者几步去尝试新的子空间。

因此我们也发现，回溯法相关问题是一类图的子问题。

Q&A:

1. 为什么是 DFS?

因为解空间的树的高度实际上是全局需要的步数，宽度是每步的可能性。
满足全局要求的解一定要能够走到树的底部。因此 DFS 的性能要比 BFS 强不知道多少。

2. 回溯为什么用递归来写?

   事实上可以用 Iteration，正如 DFS 也可以用 Iteration 一样。
   只不过用递归的代码更简洁一点。如果题目比较特殊，用 Iteration 写不难，也可以。

3. 回溯和 DFS 的关系?

   通过 LeetCode079 可以
   比较清晰的看出回溯和 DFS 是非常相似的。由于解空间的图本身具有树结构，而我们事实上要遍历树。那么我们显然发现 preorder 的 DFS 是不行的,inorder 是比较合适的。
   pruning 和尝试一种可能是 inorder 之前的部分。而回退到上一步的状态是 inorder 之后的部分。

4. 回溯和 Binary Tree 问题的关系?

   回溯很像加了 pruning 的 inorder binary tree traversal.

回溯模板

基于下面两题写的
[LC037](leetCode-037-Sudoku-Solver.md)
[LC1307]

核心部分是这样。可以写一些其他的helperfunction，以便于代码简单。

```
// 命可以起类似于 sodokuSolver/puzzleSolver之类的
private boolean backtracker(输入参数){
    1.边界条件，用来停止递归 一般直接 return true;
    2.特殊情况，需要以某一个规则直接进入下一层递归或者return false;
    3.
    开始迭代，迭代中暗藏 return true;
    for(以某种条件开始迭代){
        1.跳过不合适的值。
        2.尝试可能合适的值。
        3.
        if( backtracker(去到下面一层)  ){  // Guard Clause
            //如果下面一层传来 true
            return true;
        }
        4.
        如果下面一层不能传来一个合适的遍历的值，才会运行到这里。
        undo 第2步，将它复原，尝试下一个迭代的值。
    }
    4.
    通过了本层迭代，还不能找到合适的值 return true，说明真不行
    直接，
    return false;
}
```

上面的情况适用于只需要一个解，或者知道只有一个解。
如果要求需要储存每个解，则可以参考 N-皇后 051 的答案。
即

```
private void backtracker(输入参数){
    1.边界条件，用来停止递归 一般直接 return; 有时应当把一个解加入解集
    2.特殊情况，需要以某一个规则直接进入下一层递归或者 return;
    3.
    开始迭代，迭代中暗藏 return true;
    for(以某种条件开始迭代){
```

```
            1.跳过不合适的值。即剪枝函数。
            2.尝试可能合适的值。
            3.
            backtracker(去到下面一层) // 这是和上一步不一样的地方
            4.
            如果下面一层不能传来一个合适的遍历的值，才会运行到这里。
            undo 第2步，将它复原，尝试下一个迭代的值。
        }
        4.
        通过了本层迭代，还不能找到合适的值，说明真不行
        直接，
        return;
    }
```