回溯（DFS）算法框架

46. 全排列

51. N 皇后

其实在本篇文章之前，已经有过关于 DFS 的总结：**排列/组合/子集 问题** 和 **秒杀所有岛屿题目（DFS）**

「DFS 算法」和「回溯算法」其实是同一个概念，本质上都是一种暴力穷举算法

「DFS 算法」其实就是在树的遍历过程中增加了「决策」。关于对树的遍历的深度理解，可以参考 **二叉树--纲领篇** 中的相关内容

框架的提出

下面先来说说 DFS 算法的核心思想，先以树为前提，便于理解。当处于回溯树的一个节点上时，你只需要思考 3 个问题：

- **路径**：已经做出的选择
- **选择列表**：当前可以做的选择
- **结束条件**：到达决策树底层，无法再做选择的条件

基于上述 3 个问题，给出相对应的伪代码：（下面的例题会详细阐述该框架）

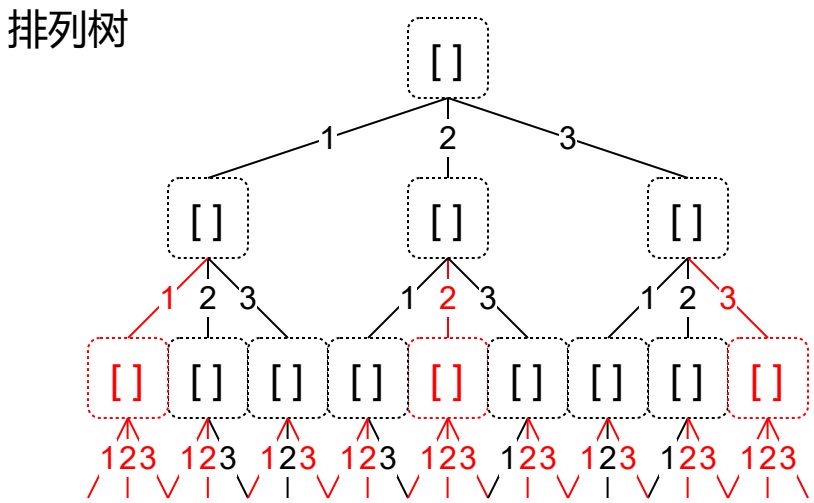
```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

全排列

题目详情可见 全排列

对于大小为 n 的全排列问题，可以抽象成一颗 n 叉树。以 3 叉树为例，如下图所示：



现在需要得到所有的全排列组合，无非就是得到所有从根节点到叶子结点的路径集合！！

问题一：由于这是一颗抽象出来的树，叶子节点并没有和树一样的特征，那如何判断到达了叶子节点？？

回答一：根据路径的长度判断是否结束！！

下面我们先不考虑其他的情况，把所有的路径遍历出来，代码如下：

```
int[] nums = {1, 2, 3};
List<Integer> path = new ArrayList<>();
List<List<Integer>> result = new ArrayList<>();
public void backtrack(int[] nums) {
    // 判断：结束条件
    if (path.size() == nums.length) {
        result.add(new ArrayList<>(path));
        return ;
    }
    for (int i = 0; i < nums.length; i++) {
        // 先序：首次进入节点，添加到「路径」中
        path.add(nums[i]);
        // 递归
        backtrack(nums);
        // 后续：即将离开节点，从「路径」中除去
        path.remove(path.size() - 1);
    }
}
```

最后运行的结果：

```
27
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 1], [1, 2, 2], [1, 2, 3], [1, 3, 1], [1, 3, 2], [1,
3, 3],
 [2, 1, 1], [2, 1, 2], [2, 1, 3], [2, 2, 1], [2, 2, 2], [2, 2, 3], [2, 3, 1], [2, 3, 2], [2,
3, 3],
 [3, 1, 1], [3, 1, 2], [3, 1, 3], [3, 2, 1], [3, 2, 2], [3, 2, 3], [3, 3, 1], [3, 3, 2], [3,
3, 3]]
```

问题二：对于上述结果，存在元素重复使用的情况，如何避免这种情况呢？？（如上图红色标注的分支均为不符合条件的情况）

回答二：新增一个 `used[]` 数组，记录元素使用情况！！

修改后的代码如下：

```
int[] nums = {1, 2, 3};
boolean[] used = new boolean[nums.length];
List<Integer> path = new ArrayList<>();
List<List<Integer>> result = new ArrayList<>();
public void backtrack(int[] nums) {
    // 判断：结束条件
    if (path.size() == nums.length) {
        result.add(new ArrayList<>(path));
        return ;
    }
    for (int i = 0; i < nums.length; i++) {
        // 如果使用过了，则直接跳过
        if (used[i]) continue;
        // 先序：首次进入节点，添加到「路径」中
        path.add(nums[i]);
        // 标记使用
        used[i] = true;
        // 递归
        backtrack(nums);
        // 后续：即将离开节点，从「路径」中除去
        path.remove(path.size() - 1);
        // 取消标记
        used[i] = false;
    }
}
```

最后运行结果如下：

```
6
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

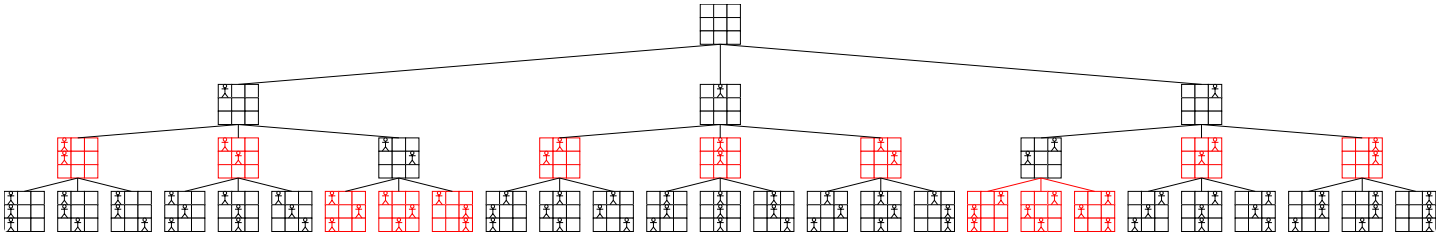
N 皇后

题目详情可见 N 皇后

首先按照框架的思路来分析一下整个过程

对于每一行，可以有 n 个选择，即任选一个格放下棋子，只要满足要求即可；整个棋盘有 n 行，所以需要做 n 次这样的选择，每一行选择一个格子放下棋子

N 皇后的决策树如下图所示：



很不巧的是，n = 3 时，没有一种情况时符合的。不过这不重要！！！（如上图红色标注的分支均为不符合条件的情况）

问题一： 用什么数据来表示棋盘呢？！

回答一： 二维数组。 `.` 表示未放棋子； `Q` 表示放了棋子

问题二： 如何判断当前行某一格放下棋子是否满足要求？！

回答二： 具体代码如下：

```
private boolean isValid(char[][] board, int row, int col) {
    int n = board.length;
    // 检查列是否有冲突
    for (int i = 0; i < n; i++) {
        if (board[i][col] == 'Q') return false;
    }
    // 检查右上方是否有冲突
    for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 'Q') return false;
    }
    // 检查左下方是否有冲突
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 'Q') return false;
    }
    return true;
}
```

现在，给出核心代码：

```
private List<String> list;
private List<List<String>> result;
public List<List<String>> solveNQueens(int n) {
    this.n = n;
    list = new ArrayList<>();
    result = new ArrayList<>();
    char[][] board = new char[n][n];
    // 初始化棋盘
    for (int i = 0; i < n; i++) {
        Arrays.fill(board[i], '.');
    }
    // 从第 0 行开始
    backtrack(board, 0);
    return result;
}
```

```
private void backtrack(char[][] board, int row) {
    // 满足要求
    if (list.size() == board.length) {
        result = new ArrayList<>(list);
        return ;
    }
    for (int i = 0; i < board.length; i++) {
        // 该格不符合放棋子的条件
        if (!isValid(board, row, i)) continue;
        // 放棋子
        board[row][i] = 'Q';
        // 记录当前行的数据
        list.add(new String(board[row]));

        // 处理下一行
        backtrack(board, row + 1);

        // 移除棋子
        board[row][i] = '.';
        // 去除当前行的数据
        list.remove(list.size() - 1);
    }
}
```

岛屿问题

关于岛屿问题的详细分析可见 秒杀所有岛屿题目(DFS)

岛屿问题是一个图的问题，本质也是使用 DFS 算法。在这里只简单的解释一下是如何套用模版滴滴滴！！

先把岛屿的模版 copy 过来了！！如下所示:

```
// 递归: 「当前节点」「该做什么」「什么时候做」
// FloodFill: 如果当前位置是岛屿，则填充为海水
//   - 充当了 visited[] 的作用
private void dfs(int[][] grid, int i, int j) {
    int m = grid.length;
    int n = grid[0].length;
    // 越界检查
    if (i < 0 || i >= m || j < 0 || j >= n) return ;
    // 如果是海水
    if (grid[i][j] == 0) return ;
    // 否则: 1 → 0
    grid[i][j] = 0;
    // 递归处理上下左右
    dfs(grid, i - 1, j); // 上
    dfs(grid, i + 1, j); // 下
    dfs(grid, i, j - 1); // 左
    dfs(grid, i, j + 1); // 右
```

```
}
```

然后来分析一下这个框架的结构，看是否与本文最开始给出的框架保持一致

```
// 越界检查
if (i < 0 || i ⩾ m || j < 0 || j ⩾ n) return ;
// 如果是海水
if (grid[i][j] == 0) return ;
```

上面两行代码实质是给出了结束递归的 base case。类似于「满足结束条件」时，保存结果，并返回

```
grid[i][j] = 0;
```

上面一行代码是处理当前节点的逻辑部分

```
dfs(grid, i - 1, j); // 上
dfs(grid, i + 1, j); // 下
dfs(grid, i, j - 1); // 左
dfs(grid, i, j + 1); // 右
```

这四行代码表示可以从 4 个方向进行递归搜索

至于这里为什么没有最后的撤销操作？因为需要保存修改后的结果，后续搜索过程会用到，所以这里不能撤销！！