回溯法

回溯法，又被称为“试探法”。解决问题时，每进行一步，都是抱着试试看的态度，如果发现当前选择并不是最好的，或者这么走下去肯定达不到目标，立刻做回退操作重新选择。这种走不通就回退再走的方法就是回溯法。

回溯VS递归

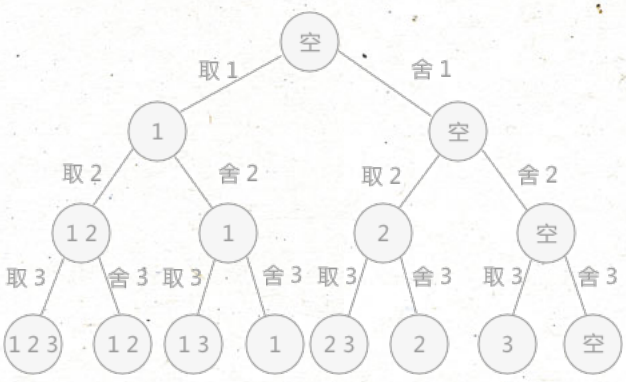
很多人认为回溯和递归是一样的，其实不然。在回溯法中可以看到有递归的身影，但是两者是有区别的。

回溯法从问题本身出发，寻找可能实现的所有情况。和穷举法的思想相近，不同在于穷举法是将所有的情况都列举出来以后再一一筛选，而回溯法在列举过程如果发现当前情况根本不可能存在，就停止后续的所有工作，返回上一步进行新的尝试。

递归是从问题的结果出发，例如求 n！，要想知道 n！的结果，就需要知道 n*(n-1)! 的结果，而要想知道 (n-1)! 结果，就需要提前知道 (n-1)*(n-2)!。这样不断地向自己提问，不断地调用自己的思想就是递归。
回溯和递归唯一的联系就是，回溯法可以用递归思想实现。

回溯法与树的遍历

使用回溯法解决问题的过程，实际上是建立一棵“状态树”的过程。例如，在解决列举集合{1,2,3}所有子集的问题中，对于每个元素，都有两种状态，取还是舍，所以构建的状态树为：

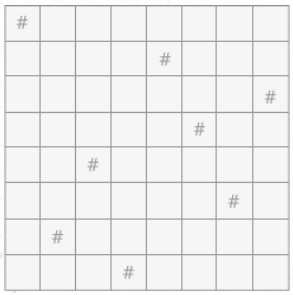


回溯法的求解过程实质上是先序遍历“状态树”的过程。树中每一个叶子结点，都有可能是问题的答案。满二叉树，得到的叶子结点全部都是问题的解。

在某些情况下，回溯法解决问题的过程中创建的状态树并不都是满二叉树，因为在试探的过程中，有时会发现此种情况下，再往下进行没有意义，所以会放弃这条死路，回溯到上一步。在树中的体现，就是在树的最后一层不是满的，即不是满二叉树，需要自己判断哪些叶子结点代表的是正确的结果。

回溯法解决八皇后问题

八皇后问题是以国际象棋为背景的问题：有八个皇后（可以当成八个棋子），如何在 8*8 的棋盘中放置八个皇后，使得任意两个皇后都不在同一条横线、纵线或者斜线上。



八皇后问题示例（#代表皇后）

八皇后问题是使用回溯法解决的典型案例。算法的解决思路是：

1. 从棋盘的第一行开始，从第一个位置开始，依次判断当前位置是否能够放置皇后，判断的依据为：同该行之前的所有行中皇后的所在位置进行比较，如果在同一列，或者在同一条斜线上（斜线有两条，为正方形的两个对角线），都不符合要求，继续检验后序的位置。
2. 如果该行所有位置都不符合要求，则回溯到前一行，改变皇后的位置，继续试探。
3. 如果试探到最后一行，所有皇后摆放完毕，则直接打印出 8*8 的棋盘。最后一定要记得将棋盘恢复原样，避免影响下一次摆放。

```
/**
 * 递归计算 N 皇后的解
 * @param {number} n
 * @param {number[]} tmp 长度为 n 的数组，tmp[i] 代表第 i 行的皇后放置的位置
 * @param {string[]} res
 */

// 解法一
function solveNQueens(n) {
  // 构建棋盘
  const board = new Array(n);
  for (let i = 0; i < n; i++) {
    board[i] = new Array(n).fill('.');
  }
  // 变量
  const res = [];
  const cols = new Set(); // 列集，记录出现过皇后的列
  const diag1 = new Set(); // 正对角线集
  const diag2 = new Set(); // 反对角线集
  // 判断是否可走
  function verify(row, col) {
    const mask1 = diag1.has(row - col);
    const mask2 = diag2.has(row + col);
    const mask3 = cols.has(col);
    return mask1 || mask2 || mask3;
  }
  //缓存
  function setCache(row, col) {
    diag1.add(row - col);
    diag2.add(row + col);
    cols.add(col);
  }
  //清除缓存
  function delCache(row, col) {
    diag1.delete(row - col);
    diag2.delete(row + col);
    cols.delete(col);
  }

```

```
  function helper(row) {
    if (row === n) {
      const stringsBoard = board.slice(); // 拷贝一份board
      for (let i = 0; i < n; i++) {
        stringsBoard[i] = stringsBoard[i].join(''); // 将每一行拼成字符串
      }
      res.push(stringsBoard)
      return
    }
    for (let col = 0; col < n; col++) {
      if (!verify(row, col)) { // 满足条件
        board[row][col] = 'Q';
        setCache(row, col);
        helper(row + 1);
        board[row][col] = '.';
        delCache(row, col)
      }
    }
  }
  helper(0); // 从第0行开始放置
  return res;
}

solveNQueens(4);


// 解法二
/**
 * 递归计算 N 皇后的解
 * @param {number} n
 * @param {number[]} tmp 长度为 n 的数组，tmp[i] 代表第 i 行的皇后放置的位置
 * @param {string[]} res
 */

// tmp 数组 存每层对应的正确索引 res结果
function dfs(n, tmp, res) {
  // 如果 tmp 长度为 n，代表所有皇后放置完毕
  if (tmp.length === n) {
    // 把这种解记录下来
    res.push(
      tmp.map(i => {
        let strArr = Array(n).fill('.')
        strArr.splice(i, 1, 'Q')
        return strArr.join('')
      })
    )
    return
```

```
  }
  // 每次有 n 个选择，该次放置在第几列
  for (let j = 0; j < n; j++) {
    // 如果当前列满足条件
    if (isValid(tmp, j)) {
      // 记录当前选择
      tmp.push(j)
      // 继续下一次的递归
      dfs(n, tmp, res)
      // 撤销当前选择
      tmp.pop()
    }
  }
}

function isValid(tmp, j) {
  let i = tmp.length
  for (let x = 0; x < i; x++) {
    let y = tmp[x]
    if (y === j || x - y === i - j || x + y === i + j) {
      return false
    }
  }
  return true
}
```