递归和回溯 / Recursion & Backtracking

递归的基本性质：函数调用本身

- 把大规模的问题不断地变小，再进行推导的过程

回溯：利用递归的性质

- 从问题的起始点出发，不断尝试
- 返回一步甚至多步再做选择，直到抵达终点的过程

递归 / Recursion

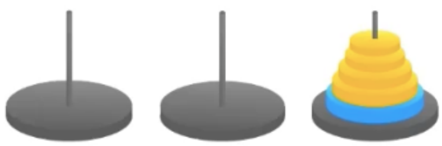
递归算法是一种调用自身函数的算法

特点：可以使一个看似复杂的问题变得简洁和易于理解

经典案例：汉诺塔（又称河内塔）

代码实现

```
void hano(char A, char B, char C, int n) {
  if (n > 0) {
    hano(A, C, B, n - 1);
    print(A + "->" + C);
    hano(B, A, C, n - 1);
  }
}
```



递归 / Recursion

算法思想

- 要懂得如何将一个问题的规模变小
- 再利用从小规模问题中得出的结果
- 结合当前的值或者情况，得出最终的结果

通俗理解

- 把要实现的递归函数，看成已经实现好的
- 直接利用解决一些子问题
- 思考：如何根据子问题的解以及当前面对的情况得出答案

自顶向下（Top-Down）

91. 解码方法

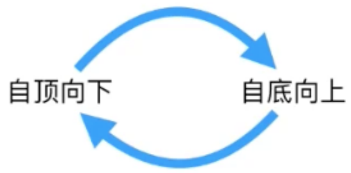
一条包含字母 `A-Z` 的消息通过以下方式进行了编码：

```
'A' -> 1
'B' -> 2
…
'Z' -> 26
```

给定一个只包含数字的非空字符串，请计算解码方法的总数。

递归 / Recursion

```
int numDecodings(String s) {
  char[] chars = s.toCharArray();
  return decode(chars, chars.length - 1);
}

int decode(char[] chars, int index) {
  if (index <= 0) {
    return 1;
  }

  int count = 0;

  char curr = chars[index];
  char prev = chars[index - 1];

  if (curr > '0') {
    count = decode(chars, index - 1);
  }

  if (prev == '1' || (prev == '2' && curr <= '6')) {
    count += decode(chars, index - 2);
  }

  return count;
}
```



- 动态规划就是典型的自底向上算法

递归写法结构总结：

```
function fn(n) {
  // 第一步：判断输入或者状态是否非法？
  if (input/state is invalid) {
    return;
  }

  // 第二步：判读递归是否应当结束？
  if (match condition) {
    return some value;
  }

  // 第三步：缩小问题规模
  result1 = fn(n1)
  result2 = fn(n2)
  ...

  // 第四步：整合结果
  return combine(result1, result2)
}
```

- 判断当前情况是否非法，如果非法就立即返回，也称为完整性检查（Sanity Check）
- 判断是否满足结束递归的条件
- 将问题的规模缩小，递归调用
- 利用在小规模问题中的答案，结合当前的数据进行整合，得出最终的答案

247. 中心对称数 II

中心对称数是指一个数字在旋转了 180 度之后看起来依旧相同的数字（或者上下颠倒地看）。

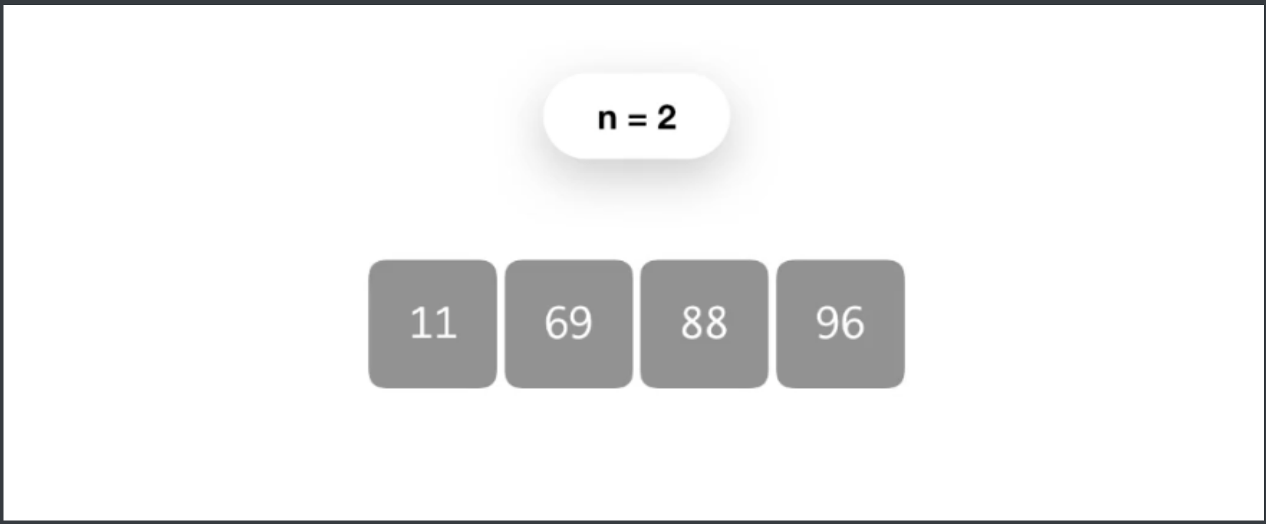
找到所有长度为 n 的中心对称数。

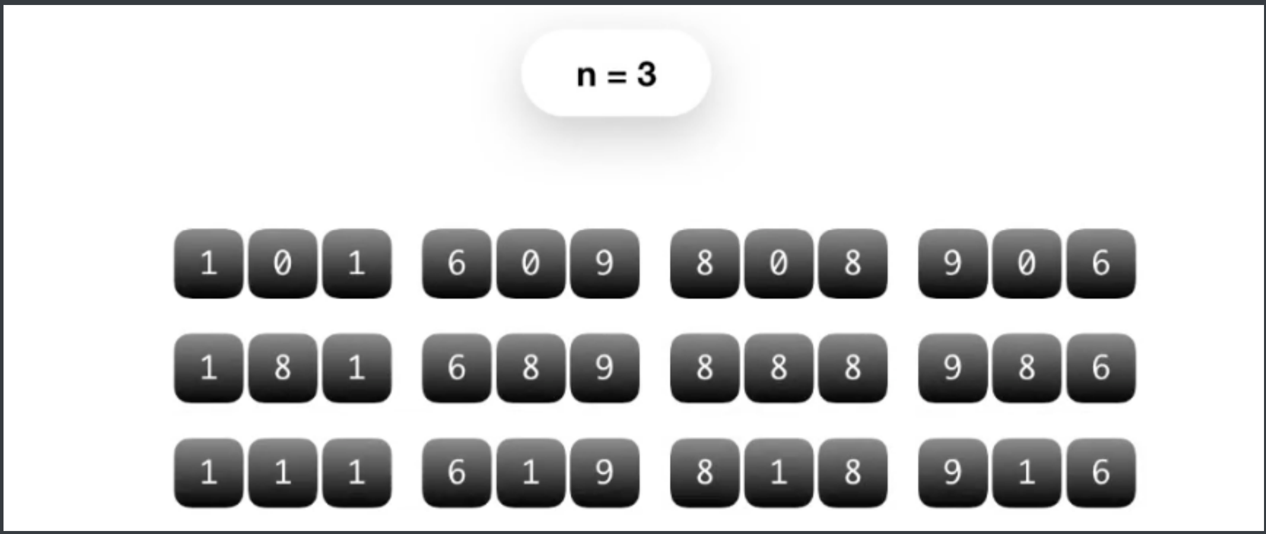
示例：
输入：n = 2
输出：["11","69","88","96"]

n = 0

n = 1

0 1 8






```java
List<String> helper(int n, int m) {
  // 第一步：判断输入或者状态是否非法?
  if (n < 0 || m < 0 || n > m) {
    throw new IllegalArgumentException("invalid input");
  }

  // 第二步：判读递归是否应当结束?
  if (n == 0) return new ArrayList<String>(Arrays.asList(""));
  if (n == 1) return new ArrayList<String>(Arrays.asList("0", "1", "8"));

  // 第三步：缩小问题规模
  List<String> list = helper(n - 2, m);

  // 第四步：整合结果
  List<String> res = new ArrayList<String>();

  for (int i = 0; i < list.size(); i++) {
    String s = list.get(i);

    if (n != m) res.add("0" + s + "0");

    res.add("1" + s + "1");
    res.add("6" + s + "9");
    res.add("8" + s + "8");
    res.add("9" + s + "6");
  }

  return res;
}
```

- 首先判断输入的值是否合法
- 当处理 n = 0 以及 n = 1 时的情况，也就是做一些递归结束条件的判读
- 将问题的规模缩小变为 n - 2
- 在 n - 2 的基础上，添加 11，69，88，96 即可

2 种递归算法解决时间复杂度分析

- 迭代法
- 公式法

迭代法

$$T(n) = 2 \times T(n-1) + O(1)$$

$\Downarrow$ 带入 $T(n)$

$$T(n) = 2\ (2 \times T(n-2) + 1) + 1 = 2^2 \times T(n-2) + (2+1)$$

$$T(n) = 2\ (2\ (2 \times T(n-3) + 1) + 1) + 1 = 2^2 \times T(n-3) + (4+2+1)$$

$$T(n) = 2\ (2\ (2\ (2 \times T(n-4) + 1) + 1) + 1) + 1 = 2^2 \times T(n-4) + (8+4+2+1)$$

$$\ldots$$

$T(n) = 2^k \times T(n-k) + (2^k - 1)$ => $T(n) = 2 \times 2^n - 1$ => O(n) = 2^n

公式法

当递归函数的时间执行函数满足如下的关系式时，可以利用公式法：

$$T(n) = a \cdot T(\frac{n}{b}) + \underline{f(n)}$$

$f(n)$ 指每次递归完毕后，额外的计算执行时间

当参数 a，b 都确定时，只看递归部分，时间复杂度就是：$O(n^{\log_b a})$

公式法 - 需要牢记的 3 种情况：

- 情况一：
  当递归部分的执行时间 $O(n^{\log_b a})$ $>$ $f(n)$ 的时候，最终的时间复杂度就是 $O(n^{\log_b a})$
- 情况二：
  当递归部分的执行时间 $O(n^{\log_b a})$ $<$ $f(n)$ 的时候，最终的时间复杂度就是 $f(n)$
- 情况三：
  当递归部分的执行时间 $O(n^{\log_b a})$ $=$ $f(n)$ 的时候，最终的时间复杂度就是 $O(n^{\log_b a})\log n$

公式法 - 示例分析

例子一：

递归排序的时间执行函数： $T(n) = 2 \times T(\frac{n}{2}) + n$

$$a = 2,\ b = 2,\ f(n) = n$$

$$log_b a = 1,\ n^1 = n$$

因此，符合第三种情况，最终的时间复杂度就是 $O(nlogn)$

公式法 - 示例分析

例子二：

```
int recursiveFn(int n) {
  if (n == 0) {
    return 0;
  }
  return recursiveFn(n / 4) + recursiveFn(n / 4);
}
```

时间执行函数： $T(n) = 2 \times T(\frac{n}{4}) + 1$

$$a = 2,\ b = 4,\ f(n) = 1$$

代入公式，得到： $n^{\log_4 2} = \sqrt{n}$

当 $n > 1$ 时，$\sqrt{n} > 1$，则时间复杂度就是：$O(\sqrt{n})$

公式法 - 示例分析

例子三：

对于第二种情况，它表示最复杂的工作发生在递归完成之后的操作，例如：

$$T(n) = 3 \times T(\frac{n}{2}) + n^2$$

$$a = 3,\ b = 2,\ f(n) = n^2$$

代入公式，得到： $n^{\log_2 3} = n^{1.48} < n^2$

回溯 / Backtracking

力扣 LeetCode

回溯算法是一种试探算法，与暴力搜索最大的区别：

在回溯算法中，是一步步向前试探，对每一步探测的情况评估，再决定是否继续，可避免走弯路

回溯算法的精华

- 出现非法的情况时，可退到之前的情景，可返回一步或多步
- 再去尝试别的路径和办法

想要采用回溯算法，就必须保证：每次都有多种尝试的可能

解决问题的套路

```
function fn(n) {
  // 第一步：判断输入或者状态是否非法?
  if (input/state is invalid) {
    return;
  }

  // 第二步：判读递归是否应当结束?
  if (match condition) {
    return some value;
  }

  // 遍历所有可能出现的情况
  for (all possible cases) {
    // 第三步：尝试下一步的可能性
    solution.push(case)
    // 递归
    result = fn(m)
    // 第四步：回溯到上一步
    solution.pop(case)
  }
}
```

- 首先判断当前情况是否非法，如果非法就立即返回
- 看看当前情况是否已经满足条件？如果是，就将当前结果保存起来并返回
- 在当前情况下，遍历所有可能出现的情况，并进行递归
- 递归完毕后，立即回溯，回溯的方法就是取消前一步进行的尝试

39. 组合总和

给定一个无重复元素的数组 *candidates* 和一个目标数 *target* ，找出 *candidates* 中所有可以使数字和为 *target* 的组合。

candidates 中的数字可以无限制重复被选取。

说明：

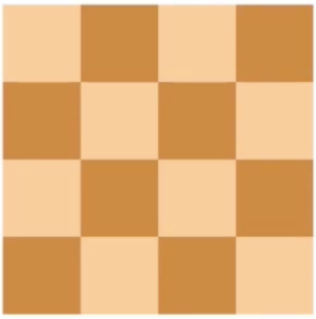
- 所有数字（包括 `target`）都是正整数。
- 解集不能包含重复的组合。

```
int[][] combinationSum(int[] candidates, int target) {
  int[][] results;
  backtracking(candidates, target, 0, [], results);
  return results;
}

void backtracking = (int[] candidates, int target, int start, int[]
solution, int[][] results) => {
  if (target < 0) {
    return;
  }

  if (target === 0) {
    results.push(solution);
    return;
  }

  for (int i = start; i < candidates.length; i++) {
    solution.push(candidates[i]);
    backtracking(candidates, target - candidates[i], i, solution,
results);
    solution.pop();
  }
}
```

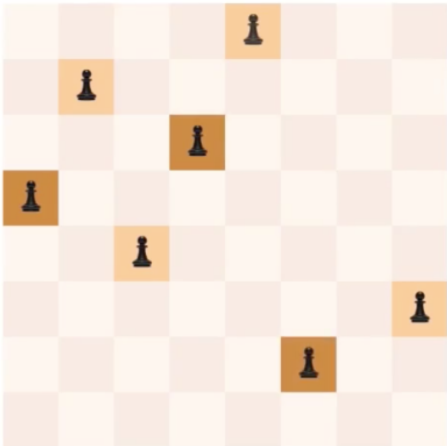
52. N 皇后 II

n 皇后问题研究的是如何将 *n* 个皇后放置在 *n*×*n* 的棋盘上，并且使皇后彼此之间不能相互攻击。

下图为 8 皇后问题的一种解法。
给定一个整数 *n*，返回 *n* 皇后不同的解决方案的数量。



```
int count;

int totalNQueens(int n) {
  count = 0;
  backtracking(n, 0, new int[n]);
  return count;
}

void backtracking(int n, int row, int[] columns) {
  // 是否在所有n行里都摆放好了皇后?
  if (row == n) {
    count++; // 找到了新的摆放方法
    return;
  }

  // 尝试着将皇后放置在当前行中的每一列
  for (int col = 0; col < n; col++) {
    columns[row] = col;
    // 检查是否合法，如果合法就继续到下一行
    if (check(row, col, columns)) {
      backtracking(n, row + 1, columns);
    }
    // 如果不合法，就不要把皇后放在这列中（回溯）
    columns[row] = -1;
  }
}
```



回溯其实是用递归实现的

因此在分析回溯的时间复杂度时，其实就是在对递归函数进行分析，方法和之前介绍的一样

分析一下 N 皇后的时间复杂度

假设 backtracking 函数的执行时间是 $T(n)$

- 首先每次都必须遍历所有的列，这里一共有 n 列

- 先要利用 check 函数检查当前的摆放方法会不会产生冲突
- 检查的时间复杂度由当前所在的行决定
- 但其上限是 n，即需要检查 n 行，所以这里的总时间复杂度就是 $O(n^2)$

分析一下 N 皇后的时间复杂度

假设 backtracking 函数的执行时间是 $T(n)$

- 首先每次都必须遍历所有的列，这里一共有 n 列
- 接下来，递归地尝试着每种摆放

- 当放好了第 1 个皇后，接下来要处理的之后 $n-1$ 个皇后
- 问题的规模减少了一个，于是执行时间变成了 $T(n-1)$

分析一下 N 皇后的时间复杂度

假设 backtracking 函数的执行时间是 $T(n)$

- 首先每次都必须遍历所有的列，这里一共有 n 列
- 接下来，递归地尝试着每种摆放
- 最终得到 $T(n)$ 的表达式：$T(n) = n \times T(n-1) + O(n^2)$

- 利用迭代法将 $T(n)$ 展开得到：
$$
\begin{aligned}
&T(n) = n \times ((n-1) \times T(n-2) + (n-1)^2 + n^2 \\
&\dots \\
&T(n) = n! + \frac{n(n+1)(2n+1)}{6} \\
&O(T(n)) = n! + O(n^3)
\end{aligned}
$$

由于 $n! > n^3$，因此，其上界就是 $n!$，即：$O(T(n)) = n!$

递归和回溯可以说是算法面试中最重要的算法考察点之一，很多其他算法都有它们的影子

- 二叉树的定义和遍历
- 归并排序、快速排序
- 动态规划
- 二分搜索

熟练掌握分析递归复杂度的方法，必须得有比较扎实的数学基础，比如要牢记等差数列、等比数列等求和公式。

- 力扣上对递归和回溯的题目分类做得很好，有丰富的题库，建议大家多做。