0119. 杨辉三角 II

ITCharge 大约 3 分钟

- 标签：数组、动态规划
- 难度：简单

题目链接

- 0119. 杨辉三角 II - 力扣

题目大意

描述：给定一个非负整数 $rowIndex$。

要求：返回杨辉三角的第 $rowIndex$ 行。

说明：

- $0 \le rowIndex \le 33$。
- 要求使用 $O(k)$ 的空间复杂度。

示例：

- 示例 1：

```py
输入：rowIndex = 3
输出：[1,3,3,1]
```

解题思路

思路 1：动态规划

因为这道题是从 0 行开始计算，则可以先将 $rowIndex$ 加 1，计算出总共的行数，即 $numRows = rowIndex + 1$。

1. 划分阶段

按照行数进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 为：杨辉三角第 i 行、第 j 列位置上的值。

3. 状态转移方程

根据观察，很容易得出状态转移方程为：$dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]$，此时 $i > 0$，$j > 0$。

4. 初始条件

- 每一行第一列都为 1，即 $dp[i][0] = 1$。
- 每一行最后一列都为 1，即 $dp[i][i] = 1$。

5. 最终结果

根据题意和状态定义，将 dp 最后一行返回。

思路 1：代码

```py
class Solution:
    def getRow(self, rowIndex: int) -> List[int]:
        # 本题从 0 行开始计算
        numRows = rowIndex + 1

        dp = [[0] * i for i in range(1, numRows + 1)]

        for i in range(numRows):
            dp[i][0] = 1
            dp[i][i] = 1

        for i in range(numRows):
```

```
            for j in range(i):
                if i != 0 and j != 0:
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]

        return dp[-1]
```

思路 1：复杂度分析

- **时间复杂度**：$O(n^2)$。初始条件赋值的时间复杂度为 $O(n)$，两重循环遍历的时间复杂度为 $O(n^2)$，所以总的时间复杂度为 $O(n^2)$。
- **空间复杂度**：$O(n^2)$。用到了二维数组保存状态，所以总体空间复杂度为 $O(n^2)$。

思路 2：动态规划 + 滚动数组优化

因为 $dp[i][j]$ 仅依赖于上一行（第 $i - 1$ 行）的 $dp[i - 1][j - 1]$ 和 $dp[i - 1][j]$，所以我们没必要保存所有阶段的状态，只需要保存上一阶段的所有状态和当前阶段的所有状态就可以了，这样使用两个一维数组分别保存相邻两个阶段的所有状态就可以实现了。

其实我们还可以进一步进行优化，即我们只需要使用一个一维数组保存上一阶段的所有状态。

定义 $dp[j]$ 为杨辉三角第 i 行第 j 列上的值。则第 $i + 1$ 行、第 j 列的值可以通过 $dp[j]$ + $dp[j - 1]$ 所得到。

这样我们就可以对这个一维数组保存的「上一阶段的所有状态值」进行逐一计算，从而获取「当前阶段的所有状态值」。

需要注意：本题在计算的时候需要从右向左依次遍历每个元素位置，这是因为如果从左向右遍历，如果当前元素 $dp[j]$ 已经更新为当前阶段第 j 列位置的状态值之后，右侧 $dp[j + 1]$ 想要更新的话，需要的是上一阶段的状态值 $dp[j]$，而此时 $dp[j]$ 已经更新了，会破坏当前阶段的状态值。而是用从左向左的顺序，则不会出现该问题。

思路 2：动态规划 + 滚动数组优化代码

py

```
class Solution:
    def getRow(self, rowIndex: int) -> List[int]:
        # 本题从 0 行开始计算
        numRows = rowIndex + 1

        dp = [1 for _ in range(numRows)]
```

```
        for i in range(numRows):
            for j in range(i - 1, -1, -1):
                if i != 0 and j != 0:
                    dp[j] = dp[j - 1] + dp[j]

        return dp
```

思路 2：复杂度分析

- **时间复杂度**：$O(n^2)$。两重循环遍历的时间复杂度为 $O(n^2)$。
- **空间复杂度**：$O(n)$。不考虑最终返回值的空间占用，则总的空间复杂度为 $O(n)$。