动态规划 one pager

MLE 算法指北

2025 年 2 月 9 日

1 动态规划的核心解题步骤

1. **识别问题是否适用 DP**：
   - **最优子结构**：问题的最优解可以由子问题的最优解推导得出。
   - **重叠子问题**：相同子问题被重复计算，适合用 DP 进行优化。
2. **定义状态变量**：找到递归关系，明确 DP 数组的含义。
3. **确定状态转移方程**：通过归纳法找到状态转移公式。
4. **确定初始条件**：找出边界情况的最优解。
5. **选择计算顺序**：
   - **自顶向下（Top-Down）**递归 + 记忆化搜索。
   - **自底向上（Bottom-Up）**迭代 + DP 数组。
6. **优化空间复杂度（可选）**：通过滚动数组或状态压缩减少内存占用。

2 动态规划的代码框架

2.1 递归 + 记忆化搜索（Top-Down Approach）

```
def dp(state, memo):
    if state 是基本情况:
        return 基本值
    if state in memo:
        return memo[state]
    memo[state] = 递归调用(dp(子状态1, memo), dp(子状态2, memo), ...)
    return memo[state]
```

2.2 迭代 + 动态规划数组（Bottom-Up Approach）

```
def dp(n):
    dp = [0] * (n + 1)
    dp[0] = 基本值
    for i in range(1, n + 1):
        dp[i] = 状态转移公式(dp[i-1], dp[i-2], ...)
    return dp[n]
```

2.3 滚动数组优化（空间优化）

```
def dp(n):
    prev, curr = 0, 1
    for i in range(2, n + 1):
        next_val = 状态转移公式(prev, curr)
        prev, curr = curr, next_val
    return curr
```

3 经典应用场景

应用场景	典型问题	时间复杂度
斐波那契数列	Fibonacci 数列	$O(n)$
线性 DP	最大子数组和、爬楼梯、最小路径和	$O(n)$
区间 DP	戳气球、石子合并	$O(n^3)$
背包问题	0-1 背包、完全背包、多重背包	$O(nW)$
子序列 DP	最长公共子序列、最长递增子序列	$O(n^2)$
划分问题	分割回文子串、正则表达式匹配	$O(n^2)$

表 1: 动态规划的经典应用场景

4 斐波那契数列

4.1 经典例题

- 斐波那契数列（LeetCode 509）
- 青蛙跳台阶（LeetCode 70）
- 兔子繁殖问题

4.2 题目解析：斐波那契数列（LeetCode 509）

状态定义：

$$F(n) = F(n-1) + F(n-2)$$

边界条件：

$$F(0) = 0, \quad F(1) = 1$$

代码实现：

```
def fibonacci(n):
    if n <= 1:
        return n
    prev, curr = 0, 1
    for _ in range(2, n + 1):
        prev, curr = curr, prev + curr
    return curr
```

时间复杂度：$O(n)$ **空间复杂度**：$O(1)$（优化后）

4.3 通用方法论

- 递推定义：$F(n) = F(n-1) + F(n-2)$
- 滚动变量优化，减少空间占用
- 斐波那契变形问题（如跳台阶问题）可采用相同思路

5 线性动态规划

线性动态规划（Linear DP）指的是**状态按照一维顺序递推**的动态规划问题，通常适用于**数组或序列问题**。常见的线性 DP 主要包括：

- **前缀型 DP**（如最大子数组和）
- **计数型 DP**（如爬楼梯、不同路径）
- **最优子结构型 DP**（如最小路径和）

5.1 线性 DP 通用解题步骤

1. **定义状态**：设 $dp[i]$ 为某个子问题的最优解或计数。
2. **确定状态转移方程**：
$$dp[i] = \max(dp[i-1] + nums[i], nums[i])$$
3. **初始化**：处理 base case，如 $dp[0] = nums[0]$。
4. **计算顺序**：自底向上迭代计算，避免重复子问题计算。
5. **优化空间复杂度**（可选）：如果只依赖前几个状态，可使用滚动数组优化至 $O(1)$。

5.2 线性 DP 经典问题

问题类型	经典问题	时间复杂度
最大连续子数组和	LeetCode 53	$O(n)$
爬楼梯	LeetCode 70	$O(n)$
最小路径和	LeetCode 64	$O(n^2)$
不同路径	LeetCode 62	$O(mn)$
整数拆分	LeetCode 343	$O(n^2)$

表 2: 线性 DP 经典问题

5.3 经典问题解析

5.3.1 最大子数组和（LeetCode 53）

题目描述：给定一个整数数组 $nums$，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

状态定义：设 $dp[i]$ 表示**以 $nums[i]$ 结尾的最大连续子数组和**：

$$dp[i] = \max(dp[i-1] + nums[i], nums[i])$$

边界条件：

$$dp[0] = nums[0]$$

代码实现：

```
def maxSubArray(nums):
    max_sum = nums[0]  # 存储最大子数组和
    curr_sum = nums[0]  # 以当前元素结尾的最大子数组和

    for i in range(1, len(nums)):
        curr_sum = max(curr_sum + nums[i], nums[i])  # 状态转移
        max_sum = max(max_sum, curr_sum)  # 更新最大值

    return max_sum
```

时间复杂度：$O(n)$　**空间复杂度**：$O(1)$（使用滚动变量优化）

5.3.2 爬楼梯（LeetCode 70）

题目描述：假设你正在爬楼梯，需要 n 阶才能到达顶部。每次可以爬 1 阶或 2 阶，求有多少种不同的方法爬到顶部。

状态定义：设 $dp[i]$ 为爬到第 i 级楼梯的方法数：

$$dp[i] = dp[i-1] + dp[i-2]$$

边界条件：

$$dp[1] = 1, \quad dp[2] = 2$$

代码实现：

```
def climbStairs(n):
    if n <= 2:
        return n
    prev, curr = 1, 2  # dp[i-2], dp[i-1]
    for _ in range(3, n + 1):
        prev, curr = curr, prev + curr  # 滚动更新
    return curr
```

时间复杂度：$O(n)$　**空间复杂度**：$O(1)$（使用滚动数组优化）

5.3.3 最小路径和（LeetCode 64）

题目描述：给定一个 $m \times n$ 的网格 $grid$，每个位置都有一个非负整数，找到一条从左上角到右下角的最小路径和（只能向下或向右移动）。

状态定义：设 $dp[i][j]$ 为到达 (i, j) 位置的最小路径和：

$$dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$$

边界条件：

$$dp[0][j] = dp[0][j-1] + grid[0][j]$$
$$dp[i][0] = dp[i-1][0] + grid[i][0]$$

代码实现：

```
def minPathSum(grid):
    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]

    # 初始化
    dp[0][0] = grid[0][0]
    for i in range(1, m):
        dp[i][0] = dp[i-1][0] + grid[i][0]
    for j in range(1, n):
        dp[0][j] = dp[0][j-1] + grid[0][j]

    # 计算 DP 值
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]

    return dp[m-1][n-1]
```

时间复杂度：$O(mn)$　**空间复杂度**：$O(mn)$（可优化至 $O(n)$）

6 区间动态规划（Interval DP）

区间动态规划（Interval DP）是一类用于**处理序列或区间的最优划分问题**的动态规划。常见的区间 DP 问题包括：

- **石子合并问题**（LeetCode 1000）
- **戳气球问题**（LeetCode 312）
- **括号匹配问题**（LeetCode 1541）
- **矩阵连乘问题**（经典 DP）

区间 DP 的特点：

- 适用于**求解子区间的最优解**，然后通过合并子区间推导出更大区间的最优解。
- 采用**双层循环**遍历区间长度，通常使用**三层循环**遍历区间端点。
- 适用于**合并问题**、**切割问题**等。

6.1 区间 DP 的通用解题思路

1. **定义状态**：
   - 设 $dp[i][j]$ 表示**区间** $[i,j]$ **的最优解**（如最小代价、最大收益等）。
2. **状态转移方程**：
   - 一般是通过**枚举分割点** k，尝试不同的切割方式：

$$dp[i][j] = \min / \max_{i \le k < j} (dp[i][k] + dp[k+1][j] + \text{合并代价})$$

3. **初始化**：
   - 单个元素的区间通常初始化为 0（如石子合并问题）。
4. **计算顺序**：
   - 采用**区间长度递增的方式**计算 DP，先计算短区间，再计算长区间。

6.2 区间 DP 经典问题

问题类型	经典问题	时间复杂度
石子合并	LeetCode 1000	$O(n^3)$
戳气球	LeetCode 312	$O(n^3)$
括号匹配	LeetCode 1541	$O(n^3)$
矩阵连乘	经典 DP	$O(n^3)$

表 3: 区间 DP 经典问题

6.3 经典问题解析

6.3.1 戳气球问题（LeetCode 312）

问题描述：给定 n 个气球，每个气球有一个分数，戳破第 i 个气球的得分为 $nums[i-1] \times nums[i] \times nums[i+1]$，求最多能获得的分数。
状态定义：设 $dp[i][j]$ 表示**戳破区间** $[i,j]$ **内所有气球能获得的最大分数**。
状态转移方程：

$$dp[i][j] = \max_{i \le k \le j} (dp[i][k-1] + nums[i-1] \cdot nums[k] \cdot nums[j+1] + dp[k+1][j])$$

代码实现：

```
def maxCoins(nums):
    nums = [1] + nums + [1]
    n = len(nums)
    dp = [[0] * n for _ in range(n)]

    for length in range(1, n-1):
        for i in range(1, n-length):
            j = i + length - 1
            for k in range(i, j + 1):
                dp[i][j] = max(dp[i][j], dp[i][k-1] + nums[i-1] * nums[k] * nums[j+1] + dp[k+1][j]

    return dp[1][n-2]
```

6.3.2 石子合并问题（LeetCode 1000）

问题描述：给定 n 堆石子，每次可以合并相邻的 K 堆，合并代价为新堆的总重量，求最小合并代价。
状态定义：设 $dp[i][j]$ 表示**合并区间** $[i,j]$ **的最小代价**。
状态转移方程：

$$dp[i][j] = \min_{i \le k < j} (dp[i][k] + dp[k+1][j] + \text{合并代价})$$

代码实现：

```
def mergeStones(stones, K):
    n = len(stones)
    if (n - 1) % (K - 1): return -1  # 无法合并

    prefix = [0] * (n + 1)
    for i in range(n):
        prefix[i + 1] = prefix[i] + stones[i]

    dp = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        dp[i][i] = 0  # 单个石堆的合并代价为 0

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            for k in range(i, j, K - 1):
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j])
            if (j - i) % (K - 1) == 0:
                dp[i][j] += prefix[j + 1] - prefix[i]

    return dp[0][n - 1]
```

6.4 区间 DP 总结

- 适用于**合并**、**切割**问题，如石子合并、矩阵连乘等。
- 采用**双层循环计算区间长度**，**三层循环枚举分割点**。
- **有时候需要加哨兵**（如戳气球问题）。

7 背包问题

背包问题是一类**组合优化问题**，核心在于**如何在有限容量的情况下选择物品，使得总价值最大或总方法数最多**。

根据不同约束条件，背包问题可以分为以下几类：

- **0-1 背包**：每个物品只能选 0 或 1 次。
- **完全背包**：每个物品可以选无限次。
- **多重背包**：每个物品最多选有限次。
- **分组背包**：物品被分成多个组，每组最多选一个。
- **多维背包**：物品有多个属性限制，如重量、体积等。
- **变种背包**：如零钱兑换、目标和、子集和问题等。

7.1 背包问题的通用解题思路

1. **定义状态**：设 $dp[i][j]$ 表示前 i 个物品在总容量不超过 j 时的最优解。
2. **状态转移方程**：
   - **0-1 背包**：
   $$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w_i] + v_i)$$
   - **完全背包**：
   $$dp[i][j] = \max(dp[i-1][j], dp[i][j-w_i] + v_i)$$
3. **初始化**：
   - **0-1 背包**：$dp[0][j] = 0$。
   - **完全背包**：如果求最小硬币数，则初始化为无穷大。
4. **计算顺序**：
   - **0-1 背包**：必须**逆序遍历**。
   - **完全背包**：必须**正序遍历**。
5. **优化空间复杂度**：使用**滚动数组**，从 $O(nW)$ 降至 $O(W)$。

7.2 背包问题分类及解析

问题类型	经典问题	时间复杂度
0-1 背包	0-1 背包问题（基本版）	$O(nW)$
完全背包	零钱兑换（LeetCode 322）	$O(nW)$
多重背包	多重背包问题	$O(nW)$
分组背包	选择一组物品	$O(nW)$
变种背包	目标和（LeetCode 494）	$O(nW)$

表 4: 背包问题的分类

7.3 经典问题解析

7.3.1 0-1 背包问题

问题描述：给定 n 个物品，每个物品有重量 w_i 和价值 v_i，背包总容量为 W，求能获得的最大总价值。

状态转移方程：
$$dp[j] = \max(dp[j], dp[j-w_i] + v_i)$$

代码实现：

```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [0] * (W + 1)

    for i in range(n):
        for j in range(W, weights[i] - 1, -1):
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])

    return dp[W]
```

7.3.2 完全背包问题

问题描述：每种物品可以被选无限次，求最大价值。
状态转移方程：

$$dp[j] = \max(dp[j], dp[j - w_i] + v_i)$$

代码实现：

```
def unboundedKnapsack(weights, values, W):
    n = len(weights)
    dp = [0] * (W + 1)

    for i in range(n):
        for j in range(weights[i], W + 1):
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])

    return dp[W]
```

7.3.3 零钱兑换（LeetCode 322）

问题描述：给定硬币面值数组 ‘coins‘ 和总金额 ‘amount‘，求凑成该金额的最少硬币数。
状态转移方程：

$$dp[j] = \min(dp[j], dp[j - coins[i]] + 1)$$

代码实现：

```
def coinChange(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    for coin in coins:
        for j in range(coin, amount + 1):
            dp[j] = min(dp[j], dp[j - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1
```

7.4 背包问题总结

- **0-1 背包**：物品只能选一次，**倒序遍历**容量。
- **完全背包**：物品可无限次选取，**正序遍历**容量。
- **零钱兑换**：初始化 ‘dp[0] = 0‘，其他初始化为无穷大。
- **空间优化**：滚动数组优化，降低空间复杂度。

8 子序列类动态规划（Subsequence DP）

子序列问题通常涉及**序列的部分元素组合**，常见子序列问题包括：

- **最长子序列**（如最长递增子序列、最长公共子序列）
- **编辑距离**（如字符串变换）
- **回文子序列**（如最长回文子序列）

子序列问题的核心在于：

- **状态转移依赖于多个子问题**（通常是 $dp[i-1][j]$ 或 $dp[i][j-1]$）
- **二维 DP 常见**（如 LCS, 编辑距离）
- **空间优化可用滚动数组**

8.1 子序列 DP 的通用解题思路

1. **定义状态**：
   - 设 $dp[i][j]$ 表示**前 i 个元素与前 j 个元素的最优解**（通常是长度或最小操作次数）。
   - 若状态仅依赖前一个状态，可使用 $dp[i]$ 进行降维优化。
2. **状态转移方程**：
   - **最长公共子序列（LCS）**：

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1, & \text{if } X[i]=Y[j] \\ \max(dp[i-1][j], dp[i][j-1]), & \text{otherwise} \end{cases}$$

   - **编辑距离**：

$$dp[i][j] = \min(dp[i-1][j-1] + \text{replace}, dp[i-1][j] + \text{delete}, dp[i][j-1] + \text{insert})$$

   - **最长递增子序列（LIS）**：

$$dp[i] = \max(dp[j]+1), \quad (j < i, \text{且} nums[j] < nums[i])$$

3. **初始化**：
   - LCS/编辑距离：初始化第一行、第一列表示空序列。
   - LIS：初始化 $dp[i]=1$（每个元素自身都是长度为 1 的递增子序列）。
4. **计算顺序**：双层循环（外层遍历字符串或数组，内层填 DP 表）。
5. **优化空间复杂度**：
   - 滚动数组降低二维 DP 的空间复杂度。
   - LIS 可用二分查找优化时间复杂度。

8.2 子序列 DP 经典问题

问题类型	经典问题	时间复杂度
最长递增子序列（LIS）	LeetCode 300	$O(n^2)$ 或 $O(n\log n)$
最长公共子序列（LCS）	LeetCode 1143	$O(mn)$
编辑距离	LeetCode 72	$O(mn)$
最长回文子序列	LeetCode 516	$O(n^2)$
俄罗斯套娃信封问题	LeetCode 354	$O(n\log n)$

表 5: 子序列 DP 经典问题

8.3 经典问题解析

8.3.1 最长递增子序列（LIS, LeetCode 300）

问题描述：给定整数数组 $nums$，找到最长递增子序列的长度。
状态定义：设 $dp[i]$ 表示**以 $nums[i]$ 结尾的最长递增子序列的长度**。
状态转移方程：

$$dp[i] = \max(dp[j] + 1), \quad \forall j < i \text{ 且} nums[j] < nums[i]$$

代码实现：

```
def lengthOfLIS(nums):
    n = len(nums)
    dp = [1] * n
    for i in range(n):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

8.3.2 最长公共子序列（LCS, LeetCode 1143）

问题描述：给定字符串 ‘text1‘ 和 ‘text2‘，返回它们的最长公共子序列长度。
状态定义：设 $dp[i][j]$ 表示 **‘text1[0:i]‘ 和 ‘text2[0:j]‘ 的 LCS 长度**。
代码实现：

```
def longestCommonSubsequence(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]
```

8.4 子序列 DP 总结

- LIS 主要用于数组，LCS/编辑距离适用于字符串。
- LIS 可用二分优化，LCS 及编辑距离用二维 DP。
- LCS 常用于比对相似性，编辑距离用于字符串修改。

9 划分类动态规划（Partition DP）

划分类动态规划（Partition DP）是一种**将问题拆分为多个子区间或子段**的动态规划方法，通常用于求解：

- **字符串分割问题**（如最小回文分割）
- **数组划分问题**（如分割数组的最大值最小）
- **子集划分问题**（如 K 组均分问题）

划分类 DP 的特点：

- 适用于**需要找到最佳分割点**的问题。
- 采用**双层循环**遍历不同的划分方式。
- 适用于**最优划分、最小代价划分、回文划分等**。

9.1 划分类 DP 的通用解题思路

1. **定义状态**：
   - 设 $dp[i]$ 表示**将前 i 个元素划分的最优解**（如最少划分次数、最小代价等）。
2. **状态转移方程**：
   - 通常通过**枚举分割点** j 来计算：

$$dp[i] = \min_{j<i}(dp[j] + \text{分割代价})$$

3. **初始化**：
   - 需要初始化 $dp[0] = 0$ 或适当的默认值。
4. **计算顺序**：
   - 采用**从小到大的顺序**计算 DP，即先计算短的划分，再计算长的划分。

9.2 划分类 DP 经典问题

问题类型	经典问题	时间复杂度
最小回文划分	LeetCode 132	$O(n^2)$
分割数组的最大值最小	LeetCode 410	$O(n^2)$
K 组均分问题	LeetCode 698	$O(2^n)$

表 6: 划分类 DP 经典问题

9.3 经典问题解析

9.3.1 最小回文分割（LeetCode 132）

问题描述：给定一个字符串 ‘s‘，请将其分割成若干个子串，使得每个子串都是回文，并返回 ** 最少分割次数 **。

状态定义：设 $dp[i]$ 表示**字符串前 i 个字符的最少分割次数**。

状态转移方程：

$$dp[i] = \min_{j<i}(dp[j] + 1) \quad \text{if } s[j:i] \text{ 是回文}$$

代码实现：

```
def minCut(s):
    n = len(s)
    dp = [float('inf')] * n
    dp[0] = 0

    def isPalindrome(l, r):
        return s[l:r+1] == s[l:r+1][::-1]

    for i in range(n):
        for j in range(i + 1):
            if isPalindrome(j, i):
                dp[i] = 0 if j == 0 else min(dp[i], dp[j-1] + 1)

    return dp[-1]
```

9.3.2 分割数组的最大值最小（LeetCode 410）

问题描述：给定一个非负整数数组 'nums' 和一个整数 'm'，将数组划分为 'm' 个连续子数组，使得**子数组的最大和最小**，求最小可能的最大子数组和。

状态定义：设 $dp[i][j]$ 表示**前 i 个元素分成 j 组的最小可能最大子数组和**。

状态转移方程：

$$dp[i][j] = \min_{k<i}(\max(dp[k][j-1], \text{sum}(nums[k:i])))$$

代码实现：

```
def splitArray(nums, m):
    n = len(nums)
    dp = [[float('inf')] * (m + 1) for _ in range(n + 1)]
    prefix = [0] * (n + 1)

    for i in range(n):
        prefix[i + 1] = prefix[i] + nums[i]

    dp[0][0] = 0
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            for k in range(i):
                dp[i][j] = min(dp[i][j], max(dp[k][j-1], prefix[i] - prefix[k]))

    return dp[n][m]
```

9.4 划分类 DP 总结

- 适用于**最优划分**、**最小代价划分**、**回文划分等**问题。
- 采用**双层循环遍历不同的划分方式**。
- 通常需要**预处理某些性质**（如回文性、前缀和）。