动态规划总结

理论知识

- 用于：解决多阶段决策问题
- 特点 1：重复子问题：因为存在大量重复子问题，才需要记录之前计算的结果
- 特点 2：最优子结构：不同规模问题之家的关系
- 特点 3：无后效性：只记录阶段结果，而不关心这个结果是怎么来的，一般而言，状态定义越仔细，就能消除后效性（理论部分较难理解，需要做题体会）
- 思路方向 1：「自顶向下」递归 + 记忆化：少数问题这样做，刚开始学习时建议这样写，慢慢过渡到「自底向上」
- 思路方向 2：「自底向上」递推求解：绝大多数问题都可以这样做，需要习惯这种思考方式

1、状态

- 状态表示了求解问题的某个阶段
- 先看看题目问的能不能作为状态
- ✪ 什么状态好转移，就用什么状态，状态定义应该为转移方便而服务

2. 状态转移方程

掌握经典的状态设置以及状态转移方程，多做题，多总结。

3、初始化

- ✪ 初始化过程同样重要，请务必重视
- 直接从语义出发定义初始化
- 最小的子问题一般都比较好想，但也有例外。有些初始化状态可能不符合语义，但是可以被后来的状态所参考
- 从状态转移方程的下标思考初始化状态，注意数组下标不能越界，或者思考是否可以通过给状态数组（矩阵）多加一行（一列），从而避免复杂的初始化讨论
- 不同定义下初始化的值不一样，这种差别是很细微的，只能通过多做问题多总结加深体会

4、输出

- 有的时候，题目要的不是最后一个状态，这一点容易被忽略

5、考虑是否可以优化空间

- 「优化空间」即「表格复用」
- 在写对之前代码的前提下，看一看状态转移的过程中，是不是有一些状态使用过了以后再也用不到，因此考虑「复用表格」以解决规模更大的问题
- 写「优化空间」的代码在一定程度上会降低代码的可读性，不易于理解，如果看到了不太好理解的代码，很可能这一版代码是优化过的，需要从未优化的版本去理解
- 必须掌握的「优化空间」的问题：「0-1 背包」（理解一维数组逆序填表的合理性）和「完全背包」（理解一维数组顺序填表的合理性）

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。**

给定一个代表每个房屋存放金额的非负整数数组，计算你 **不触动警报装置的情况下** ，一夜之内能够偷窃到的最高金额。

示例 1：

```
输入：[1,2,3,1]
输出：4
解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

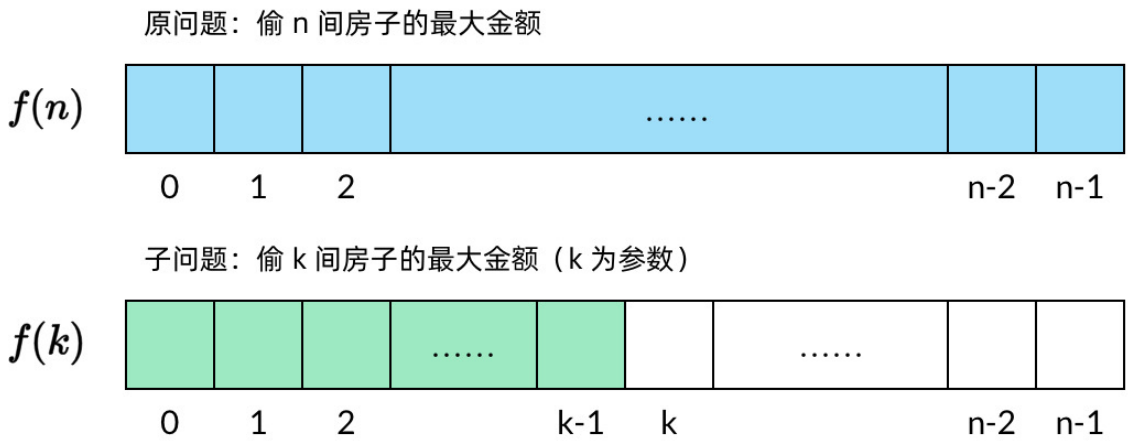
示例 2：

```
输入：[2,7,9,3,1]
输出：12
解释：偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
     偷窃到的最高金额 = 2 + 9 + 1 = 12 。
```

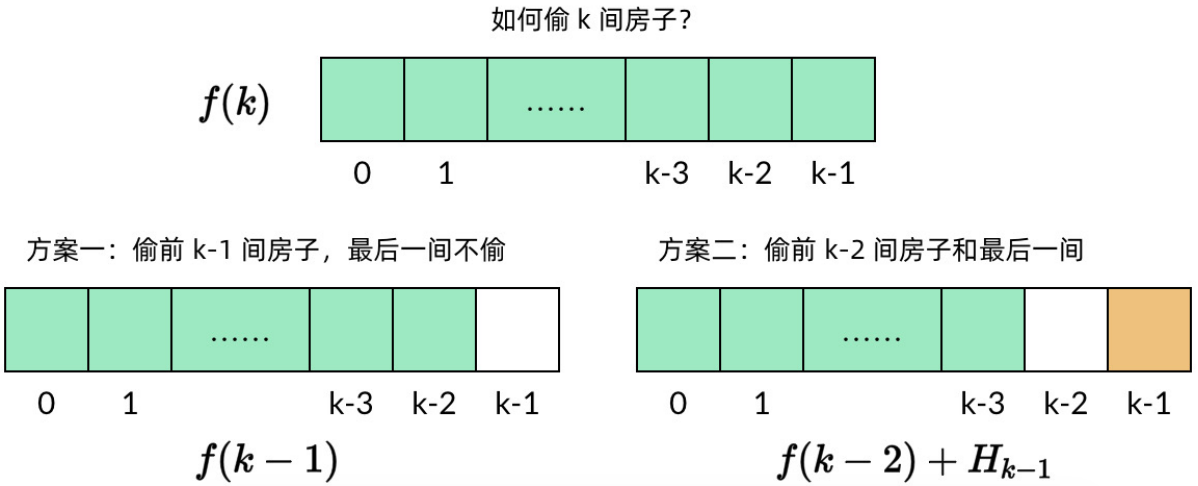
提示：

- `0 <= nums.length <= 100`
- `0 <= nums[i] <= 400`

1. 定义子问题

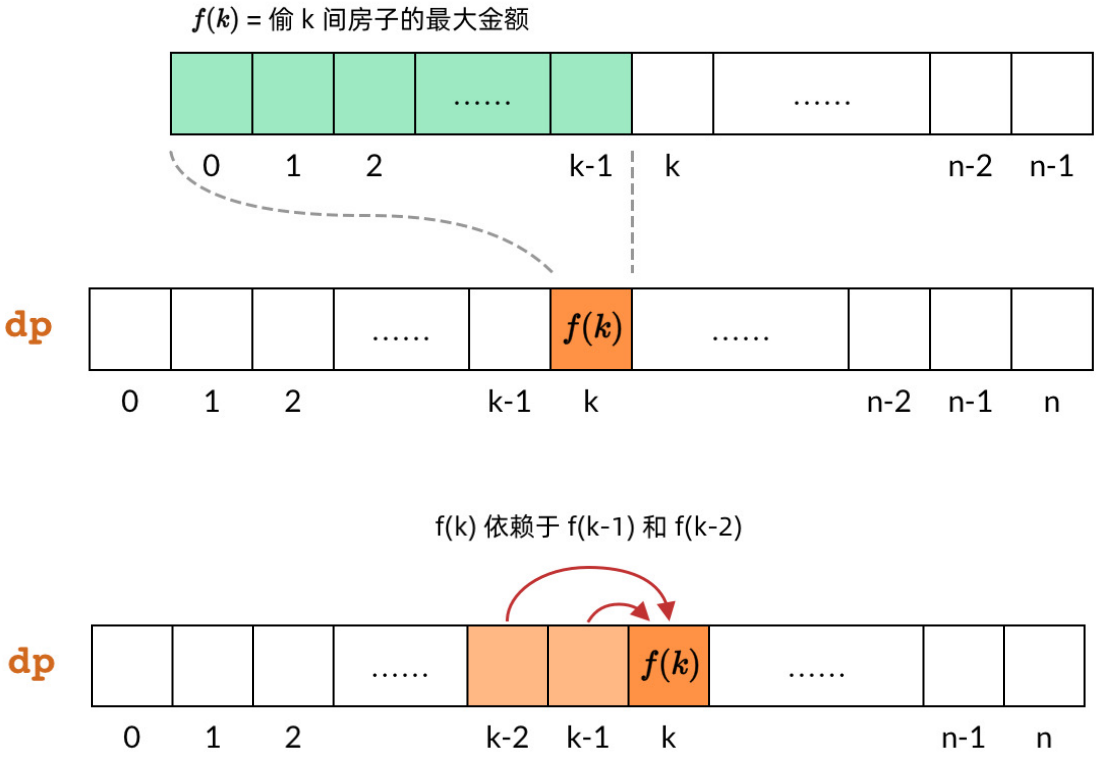


2. 写出子问题的递推关系



$$f(k) = \max\{f(k-1), H_{k-1} + f(k-2)\}$$

3. 确定DP数组的计算顺序
（自顶向下，自底向上）



3. 示例代码

传统DP代码

```
int rob(vector<int>& nums) {
    if (nums.size() == 0) {
        return 0;
    }
    // 子问题：
    // f(k) = 偷 [0..k) 房间中的最大金额

    // f(0) = 0
    // f(1) = nums[0]
    // f(k) = max{ rob(k-1), nums[k-1] + rob(k-2) }

    int N = nums.size();
    vector<int> dp(N+1, 0);
    dp[0] = 0;
    dp[1] = nums[0];
    for (int k = 2; k <= N; k++) {
        dp[k] = max(dp[k-1], nums[k-1] + dp[k-2]);
    }
    return dp[N];
}
```

空间优化DP

```
int rob(vector<int>& nums) {
    int prev = 0;
    int curr = 0;

    // 每次循环，计算“偷到当前房子为止的最大金额”
    for (int i : nums) {
        // 循环开始时，curr 表示 dp[k-1]，prev 表示 dp[k-2]
        // dp[k] = max{ dp[k-1], dp[k-2] + i }
        int temp = max(curr, prev + i);
        prev = curr;
        curr = temp;
        // 循环结束时，curr 表示 dp[k]，prev 表示 dp[k-1]
    }

    return curr;
}
```

给定一个**非空**字符串 *s* 和一个包含**非空**单词的列表 *wordDict*，判定 *s* 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
```

示例 2：

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
     注意你可以重复使用字典中的单词。
```

示例 3：

```
输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false
```

```java
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        boolean[] dp = new boolean[s.length()+1]; //默认为false
        //dp[i]表示字符串s的前i个字符组成的字符串s[0,i-1]是否能被空格拆分成一个或多个在字典中出现的单词

        dp[0] = true;
        for(int i=1; i<=s.length();i++){ //依次确定dp[i]的值
            for(int j=0; j<i; j++){
                //若dp[j]==true,且字典也包含s[j,i]这段，则dp[i]=true
                if(dp[j] && wordDict.contains(s.substring(j,i))){
                //即若字符串前j个字符都能拆分，就判断j~i能不能拆分，若字符串j~i能拆分的,则说明字符串s的前i个
                    dp[i]=true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }
}
```

```java
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> wordDictSet = new HashSet(wordDict);
        //使用HashSet查找时间复杂度为O(1) ArrayList查找时间复杂度为O(n)

        boolean[] dp = new boolean[s.length()+1];
        //dp[i]表示字符串s的前i个字符组成的字符串s[0,i-1]是否能被空格拆分成一个或多个在字典中出现的单词

        dp[0] = true;
        for(int i=1; i<=s.length();i++){ //依次判断dp[i] 从1～s.length()-1
            for(String word:wordDictSet){
                int len = word.length();
                if(i>=len && dp[i-len]==true && word.equals(s.substring(i-len,i))){
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }
}
```

行程长度编码 是一种常用的字符串压缩方法，它将连续的相同字符（重复 2 次或更多次）替换为字符和表示字符计数的数字（行程长度）。例如，用此方法压缩字符串 `"aabccc"` ，将 `"aa"` 替换为 `"a2"` ， `"ccc"` 替换为` `"c3"` 。因此压缩后的字符串变为 `"a2bc3"` 。

注意，本问题中，压缩时没有在单个字符后附加计数 `'1'` 。

给你一个字符串 `s` 和一个整数 `k` 。你需要从字符串 `s` 中删除最多 `k` 个字符，以使 `s` 的行程长度编码长度最小。

请你返回删除最多 `k` 个字符后， `s` **行程长度编码的最小长度** 。

示例 1：

```
输入：s = "aaabcccd", k = 2
输出：4
解释：在不删除任何内容的情况下，压缩后的字符串是 "a3bc3d" ，长度为 6 。最优的方案是删除 'b' 和 'd'，这样一来，压缩后的字符串为 "a3c3" ，长度是 4 。
```

示例 2：

```
输入：s = "aabbaa", k = 2
输出：2
解释：如果删去两个 'b' 字符，那么压缩后的字符串是长度为 2 的 "a4" 。
```

示例 3：

```
输入：s = "aaaaaaaaaaa", k = 0
输出：3
解释：由于 k 等于 0 ，不能删去任何字符。压缩后的字符串是 "a11" ，长度为 3 。
```

压缩串 t 由字母和数字间隔组成：

$$t = c_1 d_1 c_2 d_2 \cdots c_m d_m$$

我们用 $f[i][j]$ 表示对于原串 s 的前 i 个字符，通过删除其中的 j 个字符，剩余的 $i-j$ 个字符可以得到的最小的压缩串的长度。为了方便对边界条件进行处理，这里的 i 和 j 都从 1 开始编号。i 的最大值为 $|s|$（原串 s 的长度），j 的最大值为 k。

> 注意这里的状态表示中，我们并不关心到底删除了哪 j 个字符。

$$\text{cost}(d_x) = \begin{cases} 1, & d_x = 1 \\ 2, & 2 \le d_x \le 9 \\ 3, & 10 \le d_x \le 99 \\ 4, & d_x = 100 \end{cases}$$

如何进行状态转移呢？我们可以考虑第 i 个字符是否被删除：

- 如果第 i 个字符被删除，那么前 $i-1$ 个字符中就有 $j-1$ 个字符被删除，状态转移方程为：

$$f[i][j] = f[i-1][j-1]$$

- 如果第 i 个字符没有被删除，那么我们考虑以该字符 $s[i]$ 为结尾的一个 (c_x, d_x) 二元组，其中 $c_x = s[i]$。我们需要在 $[1, i)$ 的范围内再选择若干个（包括零个）与 $s[i]$ 相同的字符，一起进行压缩。在选择的范围内与 $s[i]$ 不相同的字符，则会全部被删除。形式化地说，我们选择了位置

$$p_1 < p_2 < \cdots < p_{d_x - 1} < i$$

$$f[i][j] = \min_{s[i_0] = s[i]} \{ f[i_0 - 1][j - \text{diff}(i_0, i)] + \text{cost}(\text{same}(i_0, i)) \}$$

  其中 $\text{diff}(i_0, i)$ 表示 $s[i_0..i]$ 中与 $s[i]$ 不同的字符数目，$\text{same}(i_0, i)$ 表示 $s[i_0..i]$ 中与 $s[i]$ 相同的字符数目，有：

$$\text{diff}(i_0, i) + \text{same}(i_0, i) = i - i_0 + 1$$

  也就是说，我们枚举满足 $s[i_0] = s[i] = c_x$ 的 i_0，选择所有在 $[i_0, i]$ 范围内的 c_x，删除剩余的字符（此时剩余的字符均不会是 c_x）。

最终的答案即为 $f[n][0..k]$ 中的最小值，即我们可以 s 中删除最多 k 个字符。由于删除一个字符永远不会劣于保留该字符，因此实际上最终的答案就是 $f[n][k]$，即我们恰好删除 k 个字符。

```java
class Solution {
    public int getLengthOfOptimalCompression(String s, int k) {
        int n = s.length();
        int[][] f = new int[n + 1][k + 1];
        for (int i = 0; i <= n; i++) {
            Arrays.fill(f[i], Integer.MAX_VALUE >> 1);
        }
        f[0][0] = 0;
        for (int i = 1; i <= n; ++i) {
            for (int j = 0; j <= k && j <= i; ++j) {
                if (j > 0) {
                    f[i][j] = f[i - 1][j - 1];
                }
                int same = 0, diff = 0;
                for (int i0 = i; i0 >= 1 && diff <= j; --i0) {
                    if (s.charAt(i0 - 1) == s.charAt(i - 1)) {
                        ++same;
                        f[i][j] = Math.min(f[i][j], f[i0 - 1][j - diff] + calc(same));
                    } else {
                        ++diff;
                    }
                }
            }
        }

        return f[n][k];
    }
```

- **石子归并**

描述

有n堆石子排成一条直线，每堆石子有一定的重量。现在要合并这些石子成为一堆石子，但是每次只能合并相邻的两堆。每次合并需要消耗一定的体力，该体力为所合并的两堆石子的重量之和。问最少需要多少体力才能将n堆石子合并成一堆石子？

输入

输入只包含若干组数据。每组数据第一行包含一个正整数n(2<=n<=100)，表示有n堆石子。接下来一行包含n个正整数a1,a2,a3,...,an（0<ai<=100，1<=i<=n）。

输出

对应输入的数据，每行输出消耗的体力。

样例输入

2
47 95

样例输出

142

分析

我们很容易想到用贪心的想法解决，但是用贪心解题算法错误。因为不一定最小的合并在一起就可以保证最终结果是最小的。

最后合并成一对石子，是由两堆石子合并而来，不妨这样定义状态转移方程：

设F[i,j]表示从第i堆到第j堆石子数总和。

Fmin(i,j)表示将从第i堆石子合并到第j堆石子的最小的得分

$$F\min(i,j)=\min_{i\le k\le j-1}\{F\min(i,k)+F\min(k+1,j)+t[i,j]\}$$

```cpp
#include <iostream>
using namespace std;
int main(){
    int a, q[110], i, j, s[110][110], r, k, p[110][110];
    while(cin >> a)  {
        for(i=1; i<=a; i++)
            cin >> q[i];
        memset(s, 0, sizeof(s));
        for(i=1; i<=a; i++)  {
             p[i][i] = q[i];
             for(j=i+1; j<=a;j++)
                 p[i][j] = p[i][j-1] + q[j];     //求前J个石子的重量和
         }
         for(r=2; r<=a; r++){
            for(i=1; i<=a-r+1; i++)  {
                 j = i + r - 1;
                 s[i][j] = INT_MAX;
                 for(k=i; k<j; k++) {
                     if(s[i][j] > s[i][k] + s[k+1][j] + p[i][j])
                         s[i][j] = s[i][k] + s[k+1][j] + p[i][j];
                 }
            }
         }
         cout << s[1][a] << endl;
     }
     return 0;
}
```

494. 目标和

难度 中等 441 收藏 分享 切换为英文 接收动态 反馈

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 `+` 和 `-`。对于数组中的任意一个整数，你都可以从 `+` 或 `-` 中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

示例:

输入： nums: [1, 1, 1, 1, 1], S: 3
输出： 5
解释：

```
-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3
```

一共有5种方法让最终目标和为3。

定义状态

搞清楚需要输出的结果后，就可以来想办法画一个表格，也就是定义dp数组的含义。根据背包问题的经验，可以将dp[i][j]定义为从数组nums中 0 - i 的元素进行加减可以得到 j 的方法数量。

状态转移方程

搞清楚状态以后，我们就可以根据状态去考虑如何根据子问题的转移从而得到整体的解。这道题的关键不是nums[i]的选与不选，而是nums[i]是加还是减，那么我们就可以将方程定义为：

- **dp[i][j] = dp[i - 1][j - nums[i]] + dp[i - 1][j + nums[i]]**

可以理解为nums[i]这个元素我可以执行加，还可以执行减，那么我dp[i][j]的结果值就是加/减之后对应位置的和。

nums：[1,1,1,1,1] S=3

dp表格：

	-5	-4	-3	-2	-1	0	1	2	3	4	5
nums[0]	0	0	0	0	1	0	1	0	0	0	0
nums[1]	0	0	0	1	0	2	0	1	0	0	0
nums[2]	0	0	1	0	3	0	3	0	1	0	0
nums[3]	0	1	0	4	0	6	0	4	0	1	0
nums[4]	1	0	5	0	10	0	10	0	5	0	1

```java
public static int findTargetSumWays(int[] nums, int s) {
    int sum = 0;
    for (int i = 0; i < nums.length; i++) {
        sum += nums[i];
    }
    // 绝对值范围超过了sum的绝对值范围则无法得到
    if (Math.abs(s) > Math.abs(sum)) return 0;

    int len = nums.length;
    // - 0 +
    int t = sum * 2 + 1;
    int[][] dp = new int[len][t];
    // 初始化
    if (nums[0] == 0) {
        dp[0][sum] = 2;
    } else {
        dp[0][sum + nums[0]] = 1;
        dp[0][sum - nums[0]] = 1;
    }

    for (int i = 1; i < len; i++) {
        for (int j = 0; j < t; j++) {
            // 边界
            int l = (j - nums[i]) >= 0 ? j - nums[i] : 0;
            int r = (j + nums[i]) < t ? j + nums[i] : 0;
            dp[i][j] = dp[i - 1][l] + dp[i - 1][r];
        }
    }
    return dp[len - 1][sum + s];
}
```