DP算法刷题路线

力扣

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    int n;  // 输入的目标数字
    if (!(cin >> n)) return 0;  // 读取输入，失败则直接退出

    // 创建动态规划数组，大小为n+1（包含0到n），初始化为0
    // 使用long long类型防止大数溢出
    vector<long long> dp(n + 1, 0);

    dp[0] = 0;  // 基础情况：第0个值
    if (n >= 1) dp[1] = 1;  // 基础情况：第1个值

    // 动态规划循环：从第2个元素开始计算，直到第n个
    for (int i = 2; i <= n; i++) {
        // 这里需要根据具体动态规划问题填写状态转移方程
        // 例如斐波那契数列：dp[i] = dp[i-1] + dp[i-2]
        // 具体状态转移方程取决于要解决的DP问题
    }

    // 输出第n个计算结果的动态规划值
    cout << dp[n] << endl;

    return 0;  // 程序正常结束
}
```

在 C 组竞赛中，解决普通一维 DP 问题通常遵循以下逻辑：

- 确定 dp 数组及下标的含义：明确 `dp[i]` 代表什么。
  - 例子（爬楼梯）：`dp[i]` 表示爬到第 i 层楼梯共有多少种走法。
- 确定状态转移方程：这是 DP 的灵魂，即如何由已知的 `dp[j]` （$j < i$）推导出 `dp[i]`。
  - 例子：要到第 i 层，可以从第 $i-1$ 层跨 1 步，也可以从第 $i-2$ 层跨 2 步。所以 $dp[i] = dp[i-1] + dp[i-2]$。
- 初始化边界条件：确定最简单情况下的解，防止数组越界或计算逻辑中断。
  - 例子：$dp[1] = 1$（1层只有1种走法），$dp[2] = 2$（2层有 1+1 或 2 两种走法）。
- 确定遍历顺序：通常是从小到大遍历，确保计算 `dp[i]` 时，它所依赖的 `dp[i-1]` 等子问题已经计算完毕。

斐波那契数列

题目描述：

斐波那契数 （通常用 $F(n)$ 表示）形成的序列称为斐波那契数列。该数列由 `0` 和 `1` 开始，后面的每一项数字都是前面两项数字的和。也就是：

- $F(0) = 0$
- $F(1) = 1$

- $F(n) = F(n-1) + F(n-2)$，其中 $n > 1$

给定 n，请计算 $F(n)$。

示例 1：

- 输入：`n = 2`
- 输出：`1`
- 解释：$F(2) = F(1) + F(0) = 1 + 0 = 1$

示例 2：

- 输入：`n = 3`
- 输出：`2`
- 解释：$F(3) = F(2) + F(1) = 1 + 1 = 2$

示例 3：

- 输入：`n = 4`
- 输出：`3`
- 解释：$F(4) = F(3) + F(2) = 2 + 1 = 3$

提示：

- $0 \leq n \leq 30$

```cpp
class Solution {
public:
    int fib(int n) {
        if(n <=1){return n;}//边界
        vector<int> dp(n+1);//定义数组
        //初始化
        dp[0] = 0;
        dp[1] = 1;
        //循环遍历，状态转移方程
        for(int i = 2;i<=n;i++){
            dp[i] = dp[i-1] + dp[i-2];
        }
        return dp[n];
    }
};
```

DP 拆解

1. 确定 dp 数组及下标的含义：我们需要一个数组 dp 来存储计算结果。dp[i] 的定义是：第 i 个斐波那契数的值。
2. 确定状态转移方程：题目已经直接给出了公式：$dp[i] = dp[i-1] + dp[i-2]$。这代表想要知道当前数字，只需要知道它前面两个数字。
3. dp 数组如何初始化：这是递推的基石。如果没有初始值，方程就无法启动。 根据题目：`dp[0] = 0`，`dp[1] = 1`。
4. 确定遍历顺序：由于 `dp[i]` 依赖于 `dp[i-1]` 和 `dp[i-2]`，所以必须从前向后遍历（从 $i = 2$ 开始）。

爬楼梯

题目描述：

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 `1` 或 `2` 个台阶。你有多少种不同的方法可以爬到楼顶呢？

示例 1：

- 输入：`n = 2`
- 输出：`2`
- 解释：有两种方法可以爬到楼顶。
    1. 1 阶 + 1 阶
    2. 2 阶

示例 2：

- 输入：`n = 3`
- 输出：`3`
- 解释：有三种方法可以爬到楼顶。
    1. 1 阶 + 1 阶 + 1 阶
    2. 1 阶 + 2 阶
    3. 2 阶 + 1 阶

提示：

- $1 \leq n \leq 45$

```cpp
class Solution {
public:
    int climbStairs(int n) {
        if(n<=2){return n;}

        vector<int> dp(n+1);

        dp[1] = 1;
        dp[2] = 2;

        for(int i =3;i<=n;i++){
            dp[i] = dp[i-1]+dp[i-2];
        }

        return dp[n];
    }
};
```

1. 确定 dp 数组及下标的含义：定义 `dp[i]`：爬到第 i 阶楼梯，共有 `dp[i]` 种不同的方法。
2. 确定状态转移方程：怎么才能站到第 i 阶台阶上

- 情况 A：我从第 $i-1$ 阶跨 1 步上来的。
- 情况 B：我从第 $i-2$ 阶跨 2 步上来的。
- 方程：$dp[i] = dp[i-1] + dp[i-2]$

3. dp 数组如何初始化：

- dp[1] = 1：上 1 层楼梯只有 1 种方法（跨 1 步）。
- dp[2] = 2：上 2 层楼梯有 2 种方法（跨 1 步再跨 1 步，或者直接跨 2 步）。

4. 确定遍历顺序从左到右，从第 3 阶一直计算到第 n 阶。

使用最小花费爬楼梯

题目描述：

给你一个整数数组 `cost` ，其中 `cost[i]` 是从楼梯第 `i` 个台阶向上爬需要支付的费用。一旦你支付了此费用，即可选择向上爬一个或者两个台阶。

你可以选择从下标为 `0` 或下标为 `1` 的台阶开始爬楼梯。

请你计算并返回达到楼梯顶部的最低花费。

示例 1：

- 输入：`cost = [10, 15, 20]`
- 输出：`15`
- 解释：你将从下标为 `1` 的台阶开始。
  - 支付 `15` ，向上爬两个台阶，到达楼梯顶部。
  - 总花费为 `15` 。

示例 2：

- 输入：`cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`
- 输出：`6`
- 解释：你将从下标为 `0` 的台阶开始。
  - 支付 `1` ，向上爬两个台阶，到达下标为 `2` 的台阶。
  - 支付 `1` ，向上爬两个台阶，到达下标为 `4` 的台阶。
  - 支付 `1` ，向上爬两个台阶，到达下标为 `6` 的台阶。
  - 支付 `1` ，向上爬一个台阶，到达下标为 `7` 的台阶。
  - 支付 `1` ，向上爬两个台阶，到达下标为 `9` 的台阶。
  - 支付 `1` ，向上爬一个台阶，到达楼梯顶部。
  - 总花费为 `6` 。

提示：

- `2 <= cost.length <= 1000`
- `0 <= cost[i] <= 999`

```cpp
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();

        vector<int> dp(n+1);
        dp[0] = 0;
        dp[1] = 0;

        for(int i= 2;i<=n;i++){
            dp[i] = min(dp[i-1]+cost[i-1],dp[i-2]+cost[i-2]);
        }

        return dp[n];
    }
};
```

1. 确定 dp 数组及下标的含义：定义 `dp[i]`：到达第 `i` 层台阶所需要的最小花费。

> 楼顶是在所有台阶之后。如果 `cost` 长度为 n，楼顶对应的下标是 n。

2. 确定状态转移方程：要站到第 i 层，只有两个来源：

- 来源 A：从第 `i-1` 层爬 1 个台阶上来。花费为：到达 `i-1` 的最小花费 + 从 `i-1` 离开的花费，即 `dp[i-1] + cost[i-1]`。

- 来源 B：从第 `i-2` 层爬 2 个台阶上来。花费为：到达 `i-2` 的最小花费 + 从 `i-2` 离开的花费，即 `dp[i-2] + cost[i-2]`。
- 决策：我们要最省钱的，所以取两者中的最小值。
- 方程：$dp[i] = \min(dp[i-1] + cost[i-1], dp[i-2] + cost[i-2])$

3. dp 数组如何初始化：

题目说：你可以选择从下标为 `0` 或 `1` 的台阶开始。

- 这意味着站在 `0` 或 `1` 台阶上是不需要支付费用的（只有从这里向上爬才付钱）。
- `dp[0] = 0`
- `dp[1] = 0`

4. 确定遍历顺序：由于 dp[i] 依赖前两个状态，所以从前向后遍历，从 i = 2 开始。

打家劫舍

题目描述：

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，影响小偷窃取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 `nums` ，请计算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：`nums = [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：`nums = [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：

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

```cpp
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();

        vector<int> dp(n);

        if(n==0){return 0;}
        if(n==1){return nums[0];}

        dp[0] = nums[0];
        dp[1] = max(nums[0],nums[1]);

        for(int i =2;i<n;i++){
            dp[i] = max(dp[i-1],dp[i-2]+nums[i]);
        }
        return dp[n-1];
```

```
    }
};
```

1. 确定 dp 数组及下标的含义：定义 `dp[i]`：考虑下标 i（包括 i）以内的房屋，能偷窃到的最高总金额。
2. 确定状态转移方程：

当你站在第 i 间房前，你有两个选择：

- 选择 A：偷第 i 间房。
  - 因为相邻不能偷，所以你绝对不能偷第 $i-1$ 间。
  - 那么你的总金额 = 第 $i-2$ 间房及之前的最大金额 + 当前这间房的钱。
  - 即：`dp[i-2] + nums[i]`。
- 选择 B：不偷第 i 间房。
  - 既然不偷这一间，你的总金额就等于考虑前一间房时的最大金额。
  - 即：`dp[i-1]`。

结论：为了金额最大，我们取两者的最大值。

- 方程：$dp[i] = \max(dp[i-1], dp[i-2] + nums[i])$

3. dp 数组如何初始化

状态转移依赖于 $i-1$ 和 $i-2$，所以我们要初始化前两个状态：

- `dp[0]`：只有一间房，必偷，金额为 `nums[0]`。
- `dp[1]`：有两间房，只能选一间（选大的），金额为 `max(nums[0], nums[1])`。

4. 确定遍历顺序

由于 $dp[i]$ 是由前两个状态推导出来的，所以从前向后遍历，从 $i = 2$ 开始。

整数拆分

题目描述：

给定一个正整数 `n` ，将其拆分为 `k` 个 正整数 的和（ `k >= 2` ），并使这些整数的乘积最大化。

返回 你可以获得的最大乘积 。

示例 1：

- 输入：`n = 2`
- 输出：`1`
- 解释：$2 = 1 + 1, 1 \times 1 = 1$。

示例 2：

- 输入：`n = 10`
- 输出：`36`
- 解释：$10 = 3 + 3 + 4, 3 \times 3 \times 4 = 36$。

提示：

- `2 <= n <= 58`

```
class Solution {
public:
    int integerBreak(int n) {
        vector<int> dp(n+1,0);
```

```
        dp[2] = 1;

        for(int i=3;i<=n;i++){
            for(int j=1;j<i;j++){
                dp[i] = max(dp[i],max(j*(i-j),dp[i-j]*j));
            }
        }
        return dp[n];
    }
};
```

1. 确定 dp 数组及下标的含义：定义 `dp[i]`：拆分正整数 `i`，可以得到的最大乘积为 `dp[i]`。
2. 确定状态转移方程：

假设我们要拆分数字 `i`。我们可以先拆出一个正整数 `j` ($1 \le j < i$)，那么剩下的部分就是 `i - j`。 此时对于剩下的 `i - j`，有两种选择：

- 选择 A：不再继续拆分。乘积就是 $j \times (i - j)$。
- 选择 B：继续拆分。乘积就是 $j \times dp[i - j]$（即使用之前计算好的 `i - j` 的最大拆分乘积）。

我们需要在所有的 `j` 的取值中，找到那个能让乘积最大的方案。

- 方程：$dp[i] = \max_{1 \le j < i}\{\max(j \times (i - j), j \times dp[i - j])\}$

3. dp 数组如何初始化

- 根据题目，`n >= 2`。
- `dp[2] = 1`：数字 2 只能拆成 $1 + 1$，乘积为 1。
- 其他可以初始化为 0。

4. 确定遍历顺序

- 外层循环 `i`：从 3 遍历到 `n`（计算每个数字的最大拆分值）。
- 内层循环 `j`：从 1 遍历到 `i - 1`（尝试所有可能的拆分点）。

解码方法

题目描述：

一条包含字母 `A-Z` 的消息通过以下映射进行了 **编码**：

- `'1' -> 'A'`
- `'2' -> 'B'`
- ...
- `'25' -> 'Y'`
- `'26' -> 'Z'`

然而，在 **解码** 已编码的消息时，你意识到了有许多不同的方式来解码，因为有些编码被包含在其它编码当中（例如 `'2'` 和 `'5'` 与 `'25'`）。

例如，`"11106"` 可以映射为：

- `"AAJF"` ，将消息分组为 `(1, 1, 10, 6)`
- `"KJF"` ，将消息分组为 `(11, 10, 6)`
- 消息不能分组为 `(1, 11, 06)` ，因为 `"06"` 不是一个合法编码（只有 `"6"` 是合法的）。

注意，可能存在无法解码的字符串。

给你一个只含数字的 **非空** 字符串 `s` ，请计算并返回 **解码** 方法的 **总数** 。如果没有合法的方式解码整个字符串，返回 `0` 。

题目数据保证答案肯定是一个 32 位 的整数。

示例 1：

- 输入：`s = "12"`
- 输出：`2`
- 解释：它可以解码为 `"AB"` (1 2) 或者 `"L"` (12)。

示例 2：

- 输入：`s = "226"`
- 输出：`3`
- 解释：它可以解码为 `"BZ"` (2 26), "VF" (22 6), 或者 "BBF" (2 2 6)。

示例 3：

- 输入：`s = "06"`
- 输出：`0`
- 解释：`"06"` 无法映射到 `"F"` ，因为存在前导零（`"6"` 和 `"06"` 并不等价）。

提示：

- `1 <= s.length <= 100`
- `s` 只包含数字，并且可能包含前导零。

```cpp
class Solution {
public:
    int numDecodings(string s) {
        int n = s.length();
        if(n == 0 || s[0] == '0'){ return 0; }

        vector<int> dp(n + 1, 0);

        dp[0] = 1;
        dp[1] = 1;

        for(int i = 2; i <= n; i++){
            // 情况 A: 当前字符不为 '0', 可以独立解码
            if(s[i-1] != '0'){
                dp[i] += dp[i-1];
            }

            // 情况 B: 当前字符与前一个字符组合在 10-26 之间
            int twoDigits = (s[i-2] - '0') * 10 + (s[i-1] - '0');
            if (twoDigits >= 10 && twoDigits <= 26) {
                dp[i] += dp[i-2];
            }
        }

        return dp[n];
    }
};
```

1. 确定 dp 数组及下标的含义

- 定义 `dp[i]`：字符串 `s` 的前 `i` 个字符组成的子串，共有 `dp[i]` 种解码方法。
- 目标：返回 `dp[s.length()]`。

2. 确定状态转移方程

当我们考虑第 `i` 个字符（对应字符串下标 `i-1`）时，有两种迈步方式：

- 方式 A：迈 1 小步（解码当前这 1 个数字）
  - 只要 `s[i-1]` 不是 `'0'`，它就可以单独解码。
  - 此时，方法数承接前一个状态：`dp[i] += dp[i-1]`。
- 方式 B：迈 1 大步（和前一个数字组合解码）
  - 只要 `s[i-2]` 和 `s[i-1]` 组成的两位数在 `10` 到 `26` 之间。
  - 此时，方法数承接再前一个状态：`dp[i] += dp[i-2]`。

3. dp 数组如何初始化

- `dp[0] = 1`：代表空字符串只有 1 种解码方式（虽然没意义，但在递推中作为基石）。
- 首位检查：如果 `s[0] == '0'`，第一个字符就无法解码，直接返回 0。

4. 确定遍历顺序

- 从前向后，从 `i = 1` 开始遍历到 `s.length()`。

完全平方数

题目描述：

给你一个整数 `n` ，返回 *和为 `n` 的完全平方数的最少数量* 。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，`1`、`4`、`9` 和 `16` 都是完全平方数，而 `3` 和 `11` 不是。

示例 1：

- 输入：`n = 12`
- 输出：`3`
- 解释：`12 = 4 + 4 + 4`

示例 2：

- 输入：`n = 13`
- 输出：`2`
- 解释：`13 = 4 + 9`

提示：

- `1 <= n <= 10⁴`

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n+1,INT_MAX);

        dp[0] = 0;

        for(int i= 1;i<=n;i++){
            for(int j=1;j*j<=i;j++){
                dp[i] = min(dp[i],dp[i-j*j]+1);
            }
        }
        return dp[n];
    }
};
```

1. 确定 dp 数组及下标的含义

- 定义 `dp[i]`：凑齐正整数 `i` 所需的最少完全平方数的数量。

- 目标：求 `dp[n]`。

2. 确定状态转移方程

假设我们现在要凑数字 `i`，我们可以尝试减去一个比 `i` 小的完全平方数 j^2。

- 如果我选了 j^2 作为其中一个数，那么剩下的数值就是 $i - j^2$。
- 此时，凑齐 `i` 的数量就等于：凑齐 $i - j^2$ 的最少数量 + 1。
- 我们要从所有可能的 j^2 中选一个能让结果最小的。
- 方程：$dp[i] = \min_{1 \le j^2 \le i}(dp[i - j^2] + 1)$

3. dp 数组如何初始化

- `dp[0] = 0`：凑齐 0 需要 0 个数。
- 关键点：因为我们要找的是最小值 `min`，所以除了 `dp[0]` 以外，其他位置应该初始化为一个最大值（比如 `INT_MAX` 或者 `i` 本身，因为 i 最多由 i 个 1 组成）。

4. 确定遍历顺序

- 外层循环 `i`：从 1 遍历到 `n`，计算每个数的最少平方数个数。
- 内层循环 `j`：尝试所有平方数 $j^2 \le i$。

零钱兑换

题目描述：

给你一个整数数组 `coins` ，表示不同面额的硬币；以及一个整数 `amount` ，表示总金额。

计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额，返回 `-1` 。

你可以认为每种硬币的数量是无限的。

示例 1：

- 输入：`coins = [1, 2, 5], amount = 11`
- 输出：`3`
- 解释：$11 = 5 + 5 + 1$

示例 2：

- 输入：`coins = [2], amount = 3`
- 输出：`-1`

示例 3：

- 输入：`coins = [1], amount = 0`
- 输出：`0`

提示：

- `1 <= coins.length <= 12`
- `1 <= coins[i] <= 2³¹ - 1`
- `0 <= amount <= 10⁴`

```cpp
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount + 1, amount + 1);

        dp[0] =0;
```

```
        for (int i = 1; i <= amount; i++){
            for (int coin : coins){
                if (i - coin >= 0){
                    dp[i] = min(dp[i], dp[i - coin] + 1);
                }
            }
        }

        return dp[amount] > amount ? -1 : dp[amount];
    }
};
```

1. 确定 dp 数组及下标的含义

- 定义 `dp[i]`：凑足总金额为 `i` 所需的最少硬币个数。
- 目标：求 `dp[amount]`。

2. 确定状态转移方程

假设我们要凑金额 `i`，我们手里有各种面值的硬币 `coin`。

- 如果我选了一枚面值为 `coin` 的硬币，那么剩下的金额就是 `i - coin`。
- 转移逻辑：凑齐 `i` 的最少硬币数 = 凑齐 `i - coin` 的最少硬币数 + 1。
- 我们需要遍历 `coins` 数组中所有的面值，取其中的最小值。
- 方程：$dp[i] = \min(dp[i], dp[i - coin] + 1)$，前提是 $i - coin \geq 0$。

3. dp 数组如何初始化

- `dp[0] = 0`：凑齐金额 0 准需要 0 枚硬币。
- 关键点：由于求的是最小值 `min`，其他位置必须初始化为一个“不可能达到的大值”。
  - 建议使用 `amount + 1`，因为即使全是 1 元硬币，最多也只需要 `amount` 枚。
  - 尽量不直接用 `INT_MAX`，因为 `INT_MAX + 1` 会导致溢出变成负数。

4. 确定遍历顺序

- 外层循环 `i`：从 1 到 `amount`，依次计算每个金额。
- 内层循环 `j`：遍历 `coins` 数组，尝试每一枚硬币。